WORKSHOP PROCEEDINGS

File Systems

May 21 - 22, 1992
Ann Arbor, Michigan

USENIX

FILE SYSTEMS WORKSHOP PROCEEDINGS

SPRING 1992

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 U.S.A.

The price is $15 for members and $20 for non-members.

Outside the U.S.A and Canada, please add
$9 per copy for postage (via air printed matter).

ISBN 1-880446-43-X

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste.

Proceedings of the

USENIX File Systems Workshop

USENIX Association

May 21-22, 1992
Ann Arbor, Michigan

Program and Table of Contents
Workshop on File Systems

Ann Arbor, Michigan

May 21 - 22, 1992

Sponsored by the USENIX Association and the University of Michigan's Center for Information Technology Integration

Thursday, May 21, 199

Friday, May 22, 1992

Program Committee

Peter Honeyman, Program Chair
C.I.T.I., University of Michigan

Michael L. Kazar, *Transarc*
Larry McVoy, *Sun Microsystems*
Mendel Rosenblum, *Stanford University*
Liuba Shrira, *MIT*
Carol Kamm, *C.I.T.I., University of Michigan*

Alex - a Global Filesystem

Vincent Cate

vac@cs.cmu.edu

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

The Alex filesystem provides users and applications transparent read access to files in Internet anonymous FTP sites. Today there are thousands of anonymous FTP sites with a total of a few million files and roughly a terabyte of data. The standard approach to accessing these files involves logging in to the remote machine. This means that an application can not access remote files and that users do not have any of their aliases or local tools available when connected to a remote site. Users who want to use an application on a remote file must first manually make a local copy of the file. Not only is this inconvenient, it creates two more problems. First, there is no mechanism for automatically updating this local copy when the remote file changes. The users must keep track of where they get their files from and check to see if there are updates, and then fetch these. Second, many different users at the same site may have made copies of the same remote file, thus wasting disk space.

Alex addresses the problems with the above approach while maintaining compatibility with the existing FTP protocol so that the large collection of currently available files can be accessed. To get reasonable performance, long term file caching must be used. Thus consistency must be addressed. Traditional solutions to the cache consistency problem do not work in the Internet FTP domain: callbacks are not an option as the FTP protocol has no provisions for this, and polling over the Internet is slow. Fortunately, the usage of these files is also not traditional and lends itself to a new approach. Alex relaxes file cache consistency semantics, on a per file basis, and uses special caching algorithms that take into account the properties of the files and of the network. This approach allows a simple stateless filesystem to scale to the size of the Internet.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the National Science Foundation under grant number ECD-8907068. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

1. Introduction

Today it is possible to access a tremendous amount of data in FTP archives on the Internet. However, the current method of accessing this data is primitive. Users must run the FTP program and then login to the remote machine. Once they have done this, they can not use any of their local tools or aliases. Applications on a user's machine can not access files on remote machines, the user must manually copy them first. If the remote file changes the user must find out somehow and make another copy. Since each user is making copies, there may be many copies of a file at a given site. Since a user can not easily tell if their own copy of a file is out of date they may use an old version of a file. Thus the existing mechanism is cumbersome, slow, inefficient, and provides only limited access to remote data.

To address this problem, and to explore some ideas about wide area filesystems, I am designing and implementing a filesystem called **Alex**. The goal of this system is to let users access files in FTP sites all over the world just as they access files on their local system. Ideally, both the way it is used and its performance should match that of a local filesystem.

Providing transparent access to remote sites requires solutions to the problems of naming, heterogeneity and performance. While caching is the key to performance, this leads to the additional problem of cache consistency. I will describe how these problems are addressed in Alex.

The name **Alex** comes from the ancient Library of Alexandria. Alexandria gathered information from around the world into one easy to access location. Alex does an analogous thing in a very modern way.

In this paper, I describe the Alex architecture, how users interact with it, and what they need to understand to use it. I then describe the current NFS server implementation. After this I discuss related systems. Next I describe some of the useful and novel applications that can be built on top of this filesystem. I conclude with the current status of Alex, some conclusions, and my research agenda.

2. User View of Alex

One of my goals was to make Alex easy to use and think about. The hope is to provide a tool with a clean and simple abstraction.

2.1. Standard Filesystem Interface

To a user or application, Alex is just a normal filesystem. Any command that works on local files will work on Alex files. Since Alex is a filesystem, nothing needs to be recompiled and no libraries are changed. Thus, users can apply all of their existing skills and tools for using files.

2.2. Naming

The user sees a filesystem with a hierarchical global name space. At the top level (**/alex**) there are top-level Internet domains like "**edu**", "**com**", "**uk**", and "**jp**". Each component of the hostname becomes a directory name as shown in Figure 1. Then the remote path is added at the end. If the user does a '**ls /alex/edu/berkeley**" he sees some machine names such as "**ucbvax**" and "**sprite**" and some directories on **berkeley.edu**. From the "**ls**" it is not clear what is where. Alex paths do not use any special characters to delineating the host name.

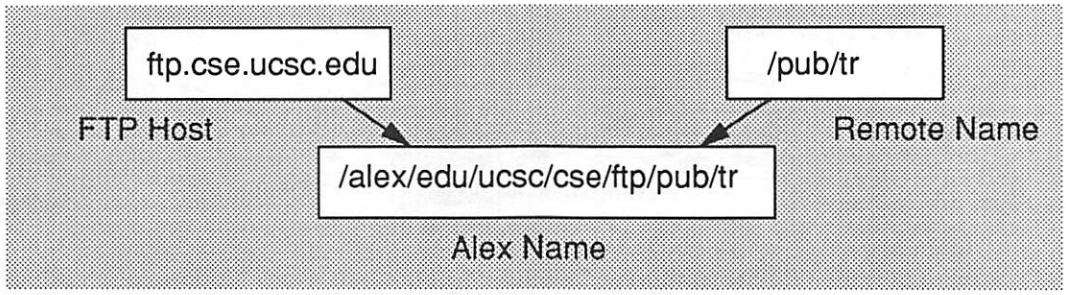


Figure 1. Example Alex Name

2.3. Control of Consistency

By default, the system guarantees to the user that the only updates that might not yet be reflected locally are ones that have happened in the last 5% of the reported age of the file. For example, if the user sees a file as being 20 hours old the system promises that the data given to the user will be at most 1 hour stale. If a file is shown as being last modified 20 days ago, the file may be up to 1 day stale. There is a maximum of 1 month and a minimum of 5 minutes. The user can override these rules and ask for an update of any directory at any time.

3. Alex Implementation

As shown in Figure 2, Alex is currently implemented as an NFS server. Alex uses the FTP protocol [Postel85] to talk to Internet FTP sites. Machines on the local area net use the NFS protocol [Nowicki89] to talk to the Alex server. NFS was chosen because it makes it trivial to add Alex to a wide range of machines. On most machines, adding Alex just involves using a simple command called mount. It is so easy, that a user who has none of my software, and is just talking with me on the phone, can start using Alex in about a minute. AFS [Howard87] scales to more clients per server but for the expected number of simultaneous users it was felt that NFS would work fine. Another motivation for choosing NFS was that it is simple and stateless. Because NFS is stateless it is fine if Alex dies and restarts (for example, during installation of a new version of the server). Since this takes only a couple of seconds, none of the clients notice.

3.1. Background

A few important characteristics of the Alex domain motivated my design choices. Key to the Alex approach is using existing FTP sites without any change to the FTP protocol. This determines a number of aspects of the domain.

FTP, like AFS [Howard87], gives us whole file transfer. The FTP protocol does not do callbacks as does AFS. Thus, Alex is fully responsible for checking the consistency of its cache with the FTP site. The FTP protocol returns directory information as a unit (not one file at a time). This, like whole file transfer, reduces the number of times you pay the network latency cost. This is a big performance win in a wide area network as big as the Internet.

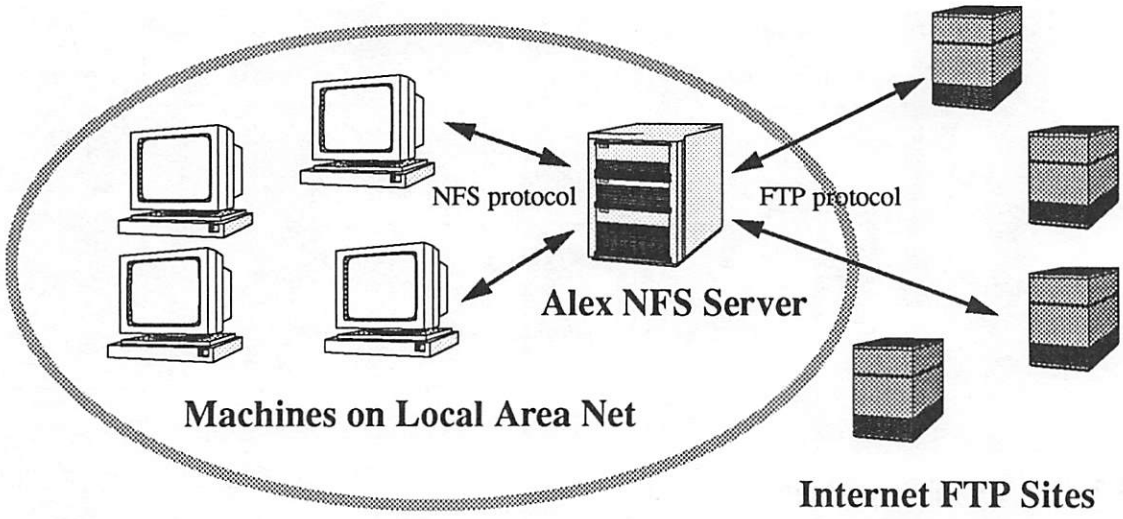


Figure 2. Alex is an NFS Server

Also important are characteristics of the domain FTP operates in. This domain is truly heterogeneous. While FTP hides some of this it also exposes plenty. Both the syntax for specifying names and the directory information returned varies among the different types of FTP sites. Another problem is that a noticeable percentage of the FTP sites are either down or inaccessible at any one time.

The type of data stored in FTP sites and the access patterns for this data are different from that of normal user files. The files change much less often than do normal files. Another important aspect of this domain is that users frequently tolerate using stale copies of files.

3.2. Naming

A path name in Alex has three parts. The first is "/alex". The second part is an Internet host name, with the most significant piece first (such as "edu") and the "."s replaced by "/"s. The third part is the path on the remote machine relative to the ftp starting directory on that machine. An example path is **/alex/edu/berkeley/pub/virus.patch**.

To resolve a name Alex first matches with the longest matching hostname it knows about, in a fashion similar to Sprite's prefix tables [Welch86]. Next, it sees if the rest of the path works. If it does not, Alex will create a possible hostname and call a nameserver. This approach seems to work almost everywhere. However, there is an ambiguity in the above name in that **pub** in is a directory on **berkeley.edu** and there is a **pub.berkeley.edu** (which both map onto the same name in Alex). Fortunately this does not happen often, in fact, pub.berkeley.edu is the only site found so far for which this does not work (and it does allow FTP anyway).

One problem with this type of naming is that a user who has set up his account to search the current directory for commands before other places such as "/bin" gets very poor performance at times. For example, if such a user wants to run "ls" while in /alex/edu/berkeley the shell will check to see if there is a /alex/edu/berkeley/ls. This causes the Alex server to check for a host called "ls.berkeley.edu" (assuming there was no "ls" file). Doing this takes a call to a remote nameserver, which is expensive. As long as they know this, users can set up their search path to avoid this problem (they should also change their path for security reasons).

3.3. Caching

To get reasonable performance a number of types of information are cached in various ways. The most important of these are listed here:

1 **Machine names** - Alex caches machine names in RAM so it can quickly resolve file names (saving calls to the nameserver).

2 **Open FTP connections** - Setting up an FTP connection takes several seconds. Once Alex has a connection it keeps it open as long as the remote site allows. Sometimes fetching a second small file from the same site is so fast that it looks as if it had been in the file cache.

3 **Type of FTP site** - Alex needs to know the type of FTP site it is talking to in order to specify names of files and directories in the appropriate syntax.

4 **Time zone information** - FTP sites report file modification times in their local timezone. NFS reports times in Greenwich time. Alex needs to know the timezone of the FTP site to make the conversion. To get the time zone of a machine may take two calls to standard Internet servers on that machine ("daytime" and "time").

5 **Directory information and files** - Copies of remote files are stored as Unix files. Directory information is parsed and the parsed information is stored as a Unix file.

6 **Failures of certain types** - Certain types of failures are cached for certain periods. If the nameserver says that it does not know of some host Alex caches that fact (since the nameserver does not cache failures). Also, if a host rejects anonymous access, Alex caches that fact. If the FTP cannot connect it caches that for a shorter period. Caching failure information is very important for good performance. Some failures are detected with timeouts, which can take a long time. Also, NFS retries can occur while waiting for a timeout from some FTP site. If the failures were not cached, each of the NFS retries would cause another long attempt to contact the FTP site.

3.4. Directories

Alex organizes directories to use disks more efficiently than a normal Unix filesystem. When a user does an "ls -F" or any operation which inspects the meta-data for each file in a directory, the Unix filesystem requires a disk seek for each file whose meta-data is not already cached in RAM. Since Alex stores all the directory information together sequentially on the disk, accesses to it are much faster. This is simple to do since there are no "hard links" to deal with. We don't actually store any details of disk allocation, so much of normal meta-data information is not needed. Inlining all file information only about doubles the directory size.

3.5. Handling Heterogeneity

There are a three ways in which heterogeneity affects Alex. These are specifying files and directory names, parsing the directory information returned, and deciding to transfer a file in binary or text mode.

Alex determines the name syntax an FTP site uses by its response to the "**pwd**" command and other commands. Once this information is known, it is straightforward to specify files in the format that the FTP site wants. Alex caches this information so that it only needs to be figured out the first time or when a problem is recognized because the site changed types (e.g. switched from VMS to Unix).

The FTP "**dir**" command returns directory information. On Unix machines this is the result of an "**ls -l**". Similar information is returned for other operating systems. Alex uses this to create all of the information for the NFS meta-data request. Alex can parse directory information from a variety of machine types (some examples are shown in Figure 3). File sizes and dates are used to determine the consistency of locally cached data. Thus, Alex, as well as the user, requires this information. Some systems, such as VMS FTP sites, give no indication of the size of a file. For these sites, Alex can only give the correct size of a file after it has been fetched. In the case of IBM VM/CMS systems, Alex can only estimate the size of a file before it is fetched (CMS provides only a maximum record size and number of records). Despite these difficulties, Alex hides the heterogeneity so well that a user can "**cd**" to a VMS machine and never know it.

DEC VMS:

```
00README.TXT;7 6 9-APR-1991 18:14 [ANONYMOUS] (RWED,RE,RE,RE)
PUB.DIR;1 8 18-OCT-1990 07:20 [ANONYMOUS] (RWED,RE,RE,RE)
```

Unix:

```
dr-xr-xr-x 3 system 320 Jul 16 14:25 0.7alpha
-rw-rw-rw- 1 system 853 Jul 16 13:41 directory
```

IBM VM/CMS

```
FUSION BIBLIO4 V 80 2219 45 11/25/91 7:00:07 FUSION
FUSION BIBLIO5 V 78 132 3 11/27/91 10:01:11 FUSION
```

Figure 3. Heterogeneous Directory Information

The decision to transfer data in "binary" or "text" format is made as follows. On Unix machines we always use binary (they all seem to use ASCII). On others, we use text unless the filename extension indicates that the file is one of a few types, such as a ".gif" file, that need to be transferred as binaries.

4. Related Work

There are many related systems. **Prospero** [Neuman89] and **Jade** [Rao91] replace libraries to create a "virtual filesystem". The problem with this approach is that applications must use these new libraries. In Prospero this means re-compiling with the new libraries. Jade replaces SunOS dynamic libraries, so applications on Suns that used dynamic libraries can access remote files via Jade. In this approach there will always be many applications that do not work with remote files. Prospero and Jade both have user specific name spaces, unlike Alex which has a single global namespace. Alex's canonical naming allows one user to say, "look around in **/alex/org/eff/ftp**" without fear of confusion.

Several applications give users easy access to remote files from within the application. The best example of this is **ange-ftp** [Norman92] which works from within GNU Emacs. **Symbolics Lisp machines** [Pearlmutter92] provided a similar function for a Lisp environment.

There are several hypertext applications that let users access files in FTP sites. Three of these are **World Wide Web** [Berner-Lee92], **Hyperbole** [Weiner91] and **HyperFTP** [Hornig90].

There are a number of systems that propagate changes to files with relaxed consistency. The **SUP** [Shafer89] and **Siphon** [Prusker91] systems are like this. Along this line, but even closer to Alex, is **Mirror** [McLoughlin91]. Mirror lets a user specify which files from a remote FTP site he wants to shadow locally.

The only other system I know of that gives access to remote FTP files via a real filesystem is **Touch** [Next92]. I have only limited information about this system. In this system there is only one site "mounted" at a time. There is no long term caching. The name space is not fixed (a user can mount the remote site anyplace he wants to). Each user runs a copy of Touch, while in Alex users mount a shared fileserver. Thus, Alex gets between user cache hits.

The **AFS** [Howard87] filesystem addresses issues of naming and performance for wide area filesystems. However, it is different in that it requires that both the server and the client be running AFS, while Alex uses existing FTP sites without modification.

5. Building on Top of Alex

While Alex nicely handles the naming and accessing issues, it does not address the indexing and organizing of the files. Since there is so much information available, this is very important. Fortunately, since it looks like a normal filesystem, Alex can easily be combined with other applications to get indexing and organization along with easy access. Systems such as WAIS [Kahle91], archie [Emtage92], Prospero [Neuman89], Gopher [Alberti91], Hyperbole [Weiner91], and World Wide Web [Berners-Lee92] can be combined with Alex to provide this functionality.

5.1. Customizing the Name Space

Symbolic links provide a user with the ability to "customize his namespace" though not with the power of some of the systems focusing on this issue. Symbolic links can be used to group related information together. For example, **/alex/edu/cmu/cs/sp/alex/links/cs-tr** is a directory with symbolic links to more than 110 directories around the world that each contain computer science technical reports (more than 3,000 papers so far). Also, symbolic links can be used to give very short names as aliases for longer ones. For example, **cs-tr** in my home directory is a symbolic link to **/alex/edu/cmu/cs/sp/alex/links/cs-tr**. A symbolic link to a remote file is as convenient as a copy and has several advantages. First, it does not take up any of the user's disk space. Next, if the remote copy changes a local copy would be out of date but the symbolic link can show the new data. Finally, the symbolic link documents where the file comes from, making it easy to go back later and look for related files.

While most symbolic links on FTP sites are modified to be relative to the machine they come from, links that point to **/alex** or **/afs** are recognized as referring to a global namespace and are not modified. This means that any FTP site can easily maintain a set of symbolic links to interesting places in Alex (as is done in the **cs-tr** directory above).

5.2. Integrating Archie, Prospero, and Alex using Archia

Archie [Emtage92] is a database that allows users to search for filenames in FTP sites. There is a client program that uses the **Prospero** [Neuman89] protocol to query the Archie database. For each match, **archie** returns a host, directory, and filename. **Archia** uses this client program and combines these 3 pieces into an Alex path. So a user can type **archia jokes** and get Alex pathnames with the word **jokes** in them. The user can then easily access these files with Alex.

5.3. Integrating Alex and WAIS

WAIS [Kahle91] stands for Wide Area Information Server. This is a software package that makes it easy to set up a database. Essentially, all you need to do is give it a list of filenames and it will index the contents. A WAIS server is run and people from around the Internet can access this database with the client software. Since Alex makes files in FTP sites look like normal files, it becomes easy to use something like WAIS to index files from all around the world. I have created two databases to experiment with this. One indexes README files, the other, computer science technical papers.

I created the **readmes** database by indexing files with names including the string "README". The initial set of pathnames was made using Archia. Since then I have used the Unix **find** command in /alex to find more. Currently I have indexed about 25,000 README files from around the world. Users can search on any words in any of these files. Since README files usually describe files in the directory they are in, this can frequently help a user find what they want.

For the **cs-techreports** database I have indexed about 3,000 computer science technical papers from around the world. I started with the above mentioned list of symbolic links to directories with technical papers. I used this list of directories and the Unix **find** command to get the filenames for the papers. I then used WAIS to index these.

6. Status

Alex is up and working. There are roughly 100 users. The server load is usually small since the average user accesses Alex for less than a half hour per week. The NFS protocol is fine for this level of load. Users have visited over 2,000 different FTP sites via Alex. Currently about 800 MB of disk is being used to cache data. With current levels of activity files usually stay in the cache more than a month past the time they are last accessed.

While real performance results are not yet available (see the Research Agenda section), I have some preliminary performance numbers. I did a "**find /alex/edu -print**" and let it run for a day. It found about 10,000 files per hour (most of which were not in the cache to start with) and had only finished a small fraction of the **edu** hosts before I killed it. When Alex was reading the 25,000 README files it averaged about 10 seconds per file (again most were not in the cache to start with). The techreports also averaged about 10 seconds per file.

The **ftpable-readmes** and **cs-techreports** database are available for use by anyone on the internet. Both receive steady use from around the world.

The source code for Alex has not yet been made available but should be this summer. Check **README** on **alex.sp.cs.cmu.edu** for current information.

7. Research Agenda

There are many research issues related to Alex. Below are some of the main ones I plan to explore.

7.1. Issues for Caching Algorithms

In this domain, there are many factors to consider when deciding what to keep in the cache and what to evict. As in other caching situations, how recently and how frequently a file has been used, as well as its size, are important factors. Additional new factors are past latency and bandwidth experienced when accessing the source for the file. The source's availability is also a factor. Thus, caching algorithms are potentially complex.

I need to determine what the important parameters are for caching algorithms and how they relate to each other. In particular, how latency, bandwidth, and availability should affect what is kept in the cache. It will be very interesting to know what sort of hit ratios we can get for the files and directories.

There are many other interesting things to measure, such as the number of times a file is used once in the cache and how long files go between usages.

7.2. Sharing of Cached Data Between Users

One of the benefits of this approach is that a cached copy of a file is frequently available because some other user has accessed the file. In [Ewing92] they found that just over half of the FTP traffic for users at one site during a short 2 week period was for files that more than one user accessed. Alex caches data for much longer periods and should get much higher levels of inter-user sharing of cached data. It will be interesting to see how often this happens. Preliminary indications are that this will be common for Alex users.

7.3. Characterization of Usage Pattern

FTP files are not typical user files, and the access patterns for FTP files are different from normal files. They are often archive files that don't ever change. Frequently the ones that do change are replaced by a new file of a slightly different name. For example **pbmplus05oct91.tar.Z** was replaced by **pbmplus10dec91.tar.Z**. This is reminiscent of immutable files in the Cedar filesystem [Gifford88].

7.4. Naming Ambiguity Problems

Among the systems visited, the only naming ambiguity noticed was **/alex/edu/berkeley/pub**. It turns out that there is nothing to ftp on **pub.berkeley.edu** so even this one instance has not been a problem. None of my users have experienced the problem. However, I plan to do a careful investigation looking for instances of this problem.

7.5. Quantifying Consistency

Alex can tell when it has given out a stale file later when it updates that directory. From this I can estimate the percentage of times users get stale data. Alex can also tell how far out of date the stale files were.

There are also cases where Alex can tell that it has stale data and clean things up. For example, if a user tries to fetch a file that no longer exists, Alex quickly updates the directory. Also if Alex fetches a file and the size is not what it was expecting Alex updates the directory information for that directory. Measuring how often these events happen also gives us some measure of consistency.

Any user can request the update of any directory. Users may desire this because they have reason to believe that the directory information is out of date (for example they saw a post saying a new file was in some

directory but it does not show up in Alex). Alex can tell if the directory has changed or not when it does this. It may also be that the user has had problems in the past and does not trust Alex to keep the files consistent. So measuring how often users do this and the percentage of time the directory really was out of date will give us two more interesting numbers to quantify Alex's cache consistency (real and user perceived).

7.6. Latency for Large Files

Right now when a user does a "**more bigfile**" where **bigfile** is not yet in the cache he does not see anything until the whole FTP has finished (since FTP is whole file transfer). This means that the user sees long latency at times. However, I will probably modify Alex so that it starts using data as soon as it comes in. Once this is done the user will see reasonable latency even for large files not in the cache.

7.7. Latency of Making FTP Connections

Setting up an FTP connection takes several seconds even on a lightly loaded machine. There are several ways that this latency might be reduced. It is possible that the FTP server code has not been tuned for fast connections, and some work in this area could improve things. Another approach is to try to reduce the number of times it is necessary to set up connections by keeping connection alive for longer periods. The problem with this approach is that it will only help a small fraction of the time, and it will increase the load on FTP sites. A third approach is to use other protocols such as NFS or Prospero on sites where they are available. However, this currently only helps with a small fraction of the FTP sites.

8. Conclusions

Alex is clearly a leap forward in accessing remote data. It makes it very easy for users and applications to access files from all around the world. Along with ease of use it also improves performance. While using FTP gives one the feel of looking at a remote place through a narrow tube, Alex gives one the feeling of actually visiting the remote place. World travel has never been easier.

This technology can be combined in a simple modular way with many different systems. Thus Alex is also an enabling technology.

This paper and other documents on Alex can be found in **/alex/edu/cmu/cs/sp/alex/doc**.

9. Bibliography

Bob Alberti, Farhad Anklesaria, Paul Lindner, Mark McCahill, Daniel Torrey, "Notes on the Internet Gopher protocol", Microcomputer and Workstation Networks Research Center, Spring 1991; Revised December 1991, /alex/edu/umn/micro/boombox/pub/gopher/gopher_protocol.

Tim Berners-Lee, Robert Cailliau, Jean-Francois Groff, Bernd Pollermann, "World-Wide Web: The Information Universe", CERN, 1211 Geneva 23, Switzerland, /alex/ch/cern/nxoc01/pub/WWW/doc/Article_9202.ps

A. Emtage, P. Deutsch, "archie - An Electronic Directory Service for the Internet", Usenix Conference Proceedings, pp. 93-110, San Francisco, CA, January 1992. /alex/ca/mcgill/cs/quiche/archie/doc/archie-usenix92-paper.ps.

David J. Ewing, Richard S. Hall, Michael F. Schwartz, "A Measurement Study of Internet File Transfer Traffic", January 1992, Department of Computer Science, University of Colorado at Boulder, CU-CS-571-92, /alex/edu/colorado/cs/bruno/pub/cs/techreports/schwartz/RD.Papers/PostScript/FTP.Meas.ps.Z.

Douglas Hornig, "HyperFTP", 1990 /alex/edu/indiana/bio/fly/util/mac/hyperftp.readme, see also /alex/edu/sdsu/ucselx/pub/mac/HyperFTP.sit.Hqx

David K. Gifford, Roger M. Needham, Michael D. Schroeder, "The Cedar File System", Communications of the ACM, Vol 31, No 3, March 1988, pp. 288-298.

Brewster Kahle, "Wide Area Information Servers", Thinking Machines, April, 1991, /alex/com/think/quake/wais/wais-discussion/overview.text.

J. Howard, M. Kazar, S. Menees,D.Nichols, M. Satyanarayana, R. Sidebothamn, M. West, "Scale and Performance in a Distributed file System", Operating Systems Review, Vol 21, No 5, November 1987. See also /alex/org/central/grand/pub/doc/afs.

B. Clifford Neuman, "The Virtual System Model for Large Distributed Operating Systems", Department of Computer Science, University of Washington, TR-89-01-07, /alex/edu/washington/cs/pub/pfs/doc/UW-CS-89-01-07.PS.Z

Next Touch, Personal communication with Beta user of Touch.

Andy Norman, "ange-ftp", /alex/edu/mit/ai/gnu/alpha/ange-ftp/ange-ftp.el.Z, see also /alex/edu/reed/pub/mailing-lists/ange-ftp.

Bill Nowicki, "NFS: Network File System Protocol Specification", Request for Comments: 1094, Sun Microsystems, March 1989, /alex/com/sri/nisc/phoebus/rfc/rfc1094.txt.

Barak Pearlmutter, Personal communication.

Francis Prusker, Edward Wobber, "The Siphon: Managing Distant Replicated Repositories", Proceedings of the workshop on Management of Replicated Data, IEEE, Nov. 1990, pp. 44-47.

J. Postel, J. Reynolds, "File Transfer Protocol (FTP)", Request for Comments: 959, /alex/com/sri/nisc/phoebus/rfc/rfc959.txt.

Herman Chung-Hwa Rao, "The Jade File System (Ph. D. Dissertation)", Department of Computer Science, University of Arizona, TR 91-18. See also "Accessing Files in an Internet: The Jade File System", /alex/edu/arizona/cs/llp/jade.ps

Steven Shafer, Mary Thompson, "The SUP Software Upgrade Protocol", School of Computer Science, Carnegie Mellon, September 1989, /alex/edu/cmu/cs/mach/sup/dup.doc.

Bob Weiner, "Hyperbole Manual - Hypertext for Everyday Work", Brown University, Dec. 1991, /alex/edu/brown/cs/wilma/pub/hyperbole/hypb.ps.Z

B. B. Welch and J. K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", Proc. of the 6th International Conference on Distributed Computing Systems, May 1986, pp. 184-189.

Lee McLoughlin, "mirror.man", Aug. 1991, /alex/uk/ac/ic/doc/src/archiving/mirror/mirror.shar.

The Prospero File System
A Global File System Based on the Virtual System Model

B. Clifford Neuman

Information Sciences Institute
University of Southern California

Distributed file systems have come into widespread use in recent years. Many allow files to be accessed over large geographic areas and across organizational boundaries. However, few systems to date have given much thought to how information should be organized in such a global environment.

This paper describes the Prospero File System, a file system based on the Virtual System Model, a model for building large systems within which users construct their own virtual systems by selecting and organizing the objects and services of interest. This customized view of a global file system makes it easier for users to keep track of files that they have identified as being of interest. The use of multiple name spaces can cause confusion. Such confusion is eliminated by support for closure: every object has an associated name space, and names specified by the object are resolved in that name space.

Tools are provided to allow views to be kept up-to-date, and to allow views to be defined as functions of other (possibly changing) views. These tools promote sharing and enable the organization of files in ways that make it easier to identify information of interest than it is in existing systems.

The prototype implementation has been used to organize information available from Internet archive sites; its directory service has been used from more than 7,500 systems in 29 countries. This paper discusses the goals of the Prospero File System, describes the prototype implementation, and discusses experience with the use of the system to date.

1 Introduction

Much attention has been paid to distributed file systems in recent years. Many of these systems allow files to be accessed over large geographic areas and across organizational boundaries. To date, however, most of the work on such systems has concentrated on access mechanisms, and less attention has been paid to the problems such environments present for the organization of information.

The Internet contains a massive amount of information, but it is hard to use that information. There are several barriers to usability: it is difficult to identify the information of interest; it is difficult to keep track of this information once found; it is difficult to share information about what is available, or to collaboratively maintain such meta-information; and the information is often scattered across multiple file systems of different types, meaning that different mechanisms are needed to access different information.

This research began as the author's dissertation at the University of Washington. It has been supported in part by the National Science Foundation (Grant No. CCR-8619663), the Washington Technology Center, Digital Equipment Corporation, and the Defense Advance Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this article are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies. The author may be reached at USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, USA. Telephone +1 (310) 822-1511, email bcn@isi.edu.

This paper describes the Prospero[1] File System. Prospero is based on the Virtual System Model, a model for building large systems that allows users to organize the information and services available to them. Prospero supports customized views of a global file system, making it easier for users to keep track of files that they have identified as being of interest. In traditional systems, a customized name space would cause confusion since the same name might refer to different objects at different times. Prospero avoids this problem by supporting closure: every object has an associated name space, and names specified by the object are resolved in that name space.

Tools are provided to allow users to keep their name spaces up-to-date. These tools allow views to be defined as functions of other, possibly changing, views and improve the user's ability to organize information, making it easier to identify information of interest than it is in existing file systems.

The Prospero File System is heterogeneous; instead of providing its own methods for storing and accessing files, it relies in existing file systems for storage and supports multiple underlying access methods. Prospero is implemented as a distributed directory service that names individual files, plus a file system interface that calls the appropriate access method once a name has been resolved. The prototype supports Sun's Network File System, the Andrew File System, and the File Transfer Protocol (FTP). For FTP, the file is automatically retrieved and the locally-cached copy is then opened.

A prototype is running and has been used from more than 7,500 systems in 29 countries on six continents. Experience with this prototype has shown that the organizational flexibility provided by the Virtual System Model is useful. Initial observations have provided insight into the way that users organize and look for information when they are not constrained to use a single, monolithic name space.

Our discussion begins by describing existing distributed file system approaches, highlighting the advantages and disadvantages of each. Section 3 describes the Virtual System Model, a model for organizing large systems that allows users to organize available information and services as they see fit. Prospero is an operating system based on that model. The file system prototype is discussed in Section 4 and performance figures are provided. Section 5 discusses experience with Prospero and describes some of the ways in which it has been used to organize information on the Internet. Related work is presented in Section 6 and future plans in Section 7. Section 8 summarizes the material presented in this paper and draws conclusions.

2 Existing Systems

The naming of files in existing distributed systems falls into four categories: host-based naming, global naming, user-centered naming, and query-based naming. This section will describe systems that fall into the first three categories, and will discuss the advantages and disadvantages of each. Systems falling into the fourth category are recent and are described as related work in Section 6.

[1] From the *Tempest* by William Shakespeare. Prospero was the rightful Duke of Milan who escaped to a desert island. When his enemies were shipwrecked on the island, Prospero used his power of illusion to separate the party into groups, each of which thought they were the only survivors. Thus, he caused each group to see a different view of the world. As time went on, the shipwrecked parties slowly learned about the others, and thus, the pieces of the other views were added to their own.

2.1 Host-Based naming

Early distributed file systems employed host-based naming to identify objects (files or directories). Examples of host-based naming include FTP [17] and Sun's Network File System[2] [20]. In host-based naming, the user must know the name of the host on which an object resides in order to access it. While relatively simple to implement, host-based naming makes it difficult to organize and locate information: the first part of a file name (the host) usually has little or no relation to the topic; logically related information stored on different systems is scattered across the name space; and as these systems are implemented, it is difficult to add links that cross system boundaries (a link is a reference from a directory to an object).

A more recent system, Alex [2], addresses the latter problem by allowing files available by FTP to be named and manipulated using the syntax of local file names. This allows users to create symbolic links to files on remote systems. Unfortunately, if the name of the target of such a link changes, the link will no longer work.

Because of these problems, many users make local copies of information that they have found on the Internet out of fear that it might move, or that they might forget where it is. Often the information is not used locally, it is copied "just in case" it is later needed. Others maintain huge lists of the information available and periodically distribute the lists through electronic mail.

2.2 Global Naming

An alternative to host-based naming employed in a number of recent systems is global naming. The Andrew File System [7], Locus [24], and Sprite [13] are among the systems that employ this approach. In global naming, all names are part of a single name space, and the name of the system on which an object resides is not explicitly part of the object's name. As these systems are implemented, however, objects with similar names must usually be stored on the same system.

A global name space solves some of the problems encountered by the host-based approach. In particular, the name of the storage site is no longer part of an object's name. It is also possible to add links to objects on other systems, though as implemented, these links are symbolic: they return a new name that must be further resolved. This means that if the name of the target of such a link changes, the link will no longer work.

Unfortunately, a global name space runs into problems as a system scales, especially once the system spans administrative boundaries. Organizations, and even users, want control over a piece of the name space. This results in a name space whose top levels are often the names of organizations, and whose second levels are the names of users. Such an organization often results in logically related information being scattered across the name space.

The alternative is to organize information by topic, rather than according to the administrative structure of the system. The difficulty with this approach is that, in a large system, there will be disagreement on what topics should appear near the top of the tree, and once topics are agreed upon, there will be disagreement on which files should be included under each topic. This problem is apparent on Usenet, a worldwide distributed message service for disseminating messages on many topics. A significant share of the messages sent on Usenet

[2] In NFS, the user must specify the host on which the file resides when mounting the file system.

discuss what messages are appropriate for particular newsgroups, whether new newsgroups should be created, and what they should be called. This clearly demonstrates the problem of reaching consensus on globally shared names.

2.3 User-Centered Naming

One of the problems with large systems is that there is a huge amount of information, and much of that information is not of interest to a particular user. User-centered naming attempts to address this problem by allowing each user to choose what is to be included in his or her name space. User-centered naming is employed by Tilde [3], QuickSilver [1], Plan 9 [18], Prospero [11], Jade [15], and some object-based systems such as Amoeba [23].

The customization supported by these system is important for a number of reasons: it reduces clutter in the user's name space; it allows users to define shorter names for frequently referenced objects; and it allows users to replace entire portions of the name space with alternative views that are more appropriate for the particular user, for example, with references to nearby instances of files instead of remote ones.

The systems with global name spaces (Section 2.2) support customization by allowing users to add symbolic links that assign short names to the files that they frequently access, but those systems allow users to assign only names that appear in their own sub-hierarchies of the file system. This is not a sufficient level of customization. Users should be able to customize all parts of their name spaces. Without this ability, users must remember when they have created customized views of parts of the name space that they are not allowed to customize directly.

User-centered naming presents several problems of its own. The lack of name transparency has the potential to make sharing difficult and to cause confusion. The problem is that the same name might refer to different objects when used in different name spaces.

Another problem is that in many user-centered systems, an object, or collection of objects, must first be added to the name space before it can be accessed[3]. Adding an object often requires that the user specify a globally unique name for the object, reintroducing all of the problems associated with the global naming approach.

A final problem is that, with the exception of Prospero, systems supporting user-centered naming do not provide adequate tools for constructing derivative views as functions of existing views. In Tilde and Plan 9 part of the problem is that views are not persistent. Instead, they are constructed by a process (often using a configuration file) and they only live as long as the processes that use them.

The problems just described are addressed by Prospero and the Virtual System Model.

3 The Virtual System Model

The Virtual System Model [12] provides a framework for organizing large systems within which users construct their own *virtual systems* by selecting objects and services that are available over the network; users then treat the selected resources as a single system, ignoring those resources that were not selected. By supporting a customized view of the system, information of interest to a user is prominently located near the center of the user's

[3]Naming in these systems might be better described as user-exclusive since objects that have not been explicitly included in the name space can not be accessed.

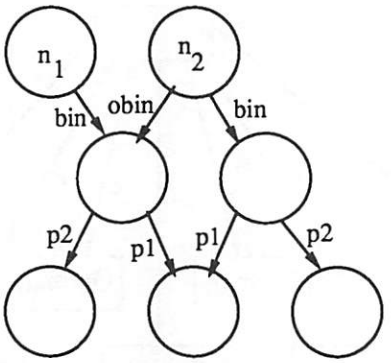


Figure 1: Two name spaces

name space, while information that is not of interest is kept out of the way. Different views also provide support for locality since users can customize what they see, replacing remote references with local ones. The Virtual System Model allows users to define views of information as functions of one or more other views. The derivative view automatically reflects any changes that occur in the views upon which it is based.

Users are able to organize the objects and information about which they know in multiple ways. The process of object discovery is facilitated by these multiple organizations, and by the fact that the information specified by users, when customizing their own name space, can be combined with other information and made available for use by others.

As indicated earlier, the fact that the same name may refer to different objects at different times can make a user-centered name space confusing and can hinder sharing. In the Virtual System Model, every object has an associated name space, forming a closure [10, 19]. In this way, the context within which a name is to be resolved can be automatically determined based on the object specifying the name. Although the same name may refer to different objects within different contexts, the correct context is always known.

3.1 Multiple Views of a Global Name Space

Within the Virtual System Model, the global naming network forms a generalized directed graph. Internal nodes in the graph represent physical directories and leaves correspond to files. The value of a directory is a collection of links, each mapping a single component of a name to a file or a directory. Each link in a physical directory is represented by a labeled edge in the graph from the node representing the physical directory to another node. Each link may have an associated function, called a filter, which when applied to the value of a directory yields a virtual directory (which defines a new view of a directory). Like a physical directory, the value of a virtual directory is a collection of links.

The Virtual System Model supports a user-centered, or more precisely, an object-centered name space. Each name space is a view of the global naming network, selected by choosing a starting node from the graph. We call the starting node the *root* of a virtual system.

Figure 1 shows a simple naming network where *n1* is the root of a name space that names two programs, */bin/p1* and */bin/p2*. A second name space, rooted at *n2*, has its own *bin* directory in which *p2* refers to a different file; the directory */bin* from *n1* has been renamed to *obin*.

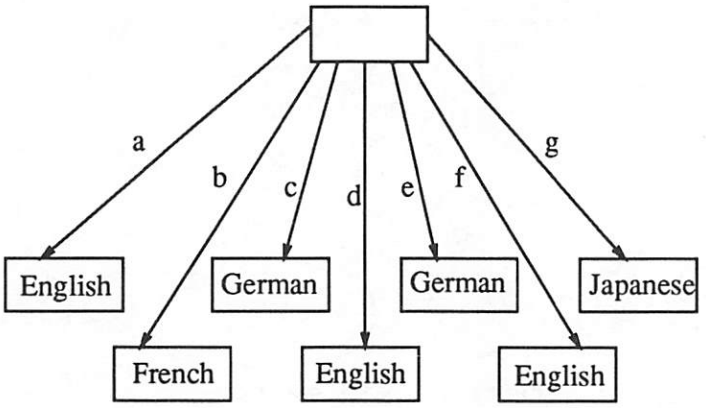


Figure 2: Directory before application of a filter

Most objects start as part of a user's name space, but with long names. As users identify information of interest, that information is moved closer to the center of the name space by adding additional links. These links are added either to a physical directory, in which case other users with views of the physical directory will see the change, or they are added to a view of a physical directory, in which case the change is visible only to other users sharing the view.

3.2 Filters and Union Links

The Virtual System Model supports customization by allowing a virtual directory to be specified as a function of other directories. This is made possible by the filter. A filter is a program, attached to a link, that allows the view of the target directory to be altered. For example, in Figure 2 files are named with the labels *a* though *g*. Associated with each file is an attribute list, one attribute of which is the language in which the text was written. The value of the language attribute is shown in the box representing the file. By attaching the **distribute()** filter to the directory link, a derived view is created within which the files appear to be distributed across subdirectories according to the value of the language attribute. The derived view is shown in Figure 3.

A filter takes the value of a directory as an argument and returns a new directory. By composing a filter with another filter already associated with a virtual directory a view can be specified as a function of another view.

Because filters are associated with links, and because the result of applying a filter is a list of links (the value of a directory) a filter can attach additional filters to the links it returns. This allows a filter to modify more than one level of the hierarchy to which it is attached. It also allows the creation of *ghost* hierarchies, parts of the name space which are specified entirely within the filter.

As described so far, views are not shared, but physical directories are. Each view of a physical directory is distinct and if a change is made to a filter that maintains a view, that change will not be visible through other views. There are cases, however, when it is desirable to share a view. For a view to be shared, the filter that implements it must appear on a link leading out of a physical directory. Unfortunately, the label on the link also names the view. If we want to share a view, but not the name of the view, we must support links that are not named.

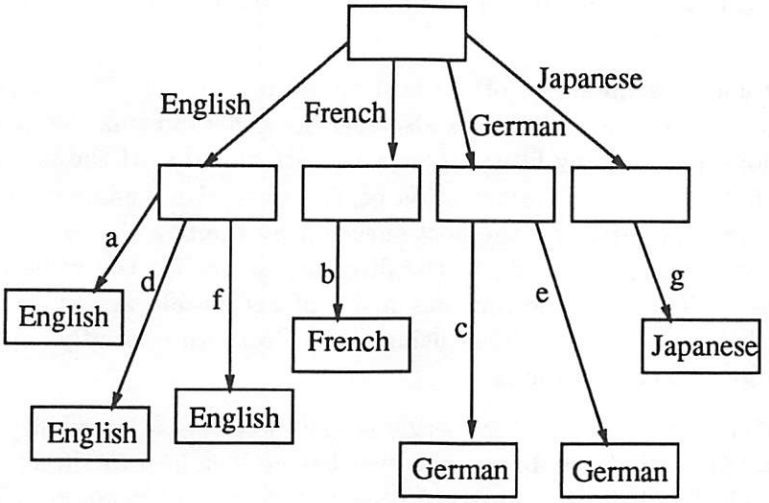


Figure 3: Directory with distribute() applied

The union link is such a link. The target of a union link is another directory. After any filters associated with a union link are applied, the result is merged with the contents of the physical directory containing the union link. Conceptually, the union link is an epsilon transition in the global naming network. Such a link is called a union link because the resulting directory appears to contain mappings that are the union of the normal links and the links from each of the directories included through union links.

The use of a union link on a physical directory containing any other links might result in more than one mapping for the same name. When resolving a name, each of the mappings must be tried[4]. As implemented, however, the union link is combined with a filter that will only pass a single mapping for each name. If an included directory contains a mapping that conflicts with one that exists in the originating directory, or from an earlier union link, then the conflicting mapping is returned separately or ignored.

Filters and union links provide a powerful mechanism for supporting customization and the manipulation of name spaces. Filters are written in standard programming languages and can take any action that can be specified in such languages. The union link allows the manipulation performed by a filter to affect the directory containing the filter.

4 The Prospero File System

The Prospero file system applies the Virtual System Model to the design of a global file system. In Prospero, files that are logically related can be grouped together, even if scattered across multiple systems. Prospero supports multiple views of the global file system and views may be defined as functions of one or more other views. A prototype of the file system has been constructed and is in use across the Internet.

4.1 Implementation

The names of files in Prospero are resolved by contacting directory servers on the hosts that store Prospero directories. The server accepts the system level name of a directory and optionally the name of the link to be returned. The server returns the links in the directory that match the specified name, or all links if the name was not specified. Attributes are

[4]This is similar to the way search paths work.

associated with objects and the directory server responds to requests for specific attributes associated with an object.

A Prospero link specifies the name of the host that stores an object, and the system level name of the object on that host. The link also specifies other information including whether the link is a union-link, and any filters associated with the link. If the target of the link is a directory, the link provides the information needed to resolve a name in that directory by contacting the directory server on the host specified by the link. To access an object that is not a directory, a request is made to the directory server for the value of the ACCESS-METHOD attribute. The response includes a list of acceptable access methods, and the information needed to access the object using each. Prospero presently supports the NFS, AFS, FTP, and local access methods.

The structure of the Prospero directory server is similar to that in capability based systems, such as Amoeba [23] in that the directory server has no idea how its directories fit into the name space. Each directory is a separate object that may be referenced by many other directories. Cycles are even allowed. The Prospero directory service is not capability-based in that the links (object handles) do not grant authorization to access the object. Additionally, links in Prospero contain information about the storage site for the object whereas in most capability based systems they provide a unique ID which must be located using a broadcast mechanism.

The Prospero client (application) remembers the system-level name (the host plus the name of the directory on that host) for the current working directory and for the root of the active virtual system. When the user or an application wishes to resolve a name, the first component is resolved relative to the appropriate directory by sending a query to the corresponding directory server. The next component is resolved by sending a query to the directory server named in the link returned by the first query. This process is repeated until all components of the name have been resolved[5]. If the named file is to be accessed, an additional query is made to obtain the access method.

Communication with the directory server is accomplished using a reliable datagram protocol implemented on top of UDP [16]. This reduces the overhead that would otherwise be incurred when establishing connections to multiple directory servers.

At any point in the resolution of a name, the directory server might return one or more union links. Such a response indicates that the directory has not been completely searched, and that the current component of the name should be resolved in the directories named in the union links[6].

If a filter is associated with a link, the filter is applied to the result of the directory query before the current component of the name is looked up[7]. A filter can remove links from or add links to a directory, change the names of links, or even change the way a directory hierarchy appears to be organized (e.g., creating subdirectories). Filters are written in C and are dynamically linked with the name resolver when they are applied.

[5] An optimization allows a directory server to resolve more than one component of the name at a time as long as all intervening directories are stored on the same server.

[6] If the directory is being listed, then the results of querying the union linked directories are merged with the rest of response from the directory that returned the union links.

[7] For this to work, the directory server must be instructed to return all links in the directory, not just those matching the component.

```
VDIR filter(dir,ip,argc,argv)
  {
    vdir_init(nd);
    sd = vlcopy(dirlink,0);
    avf = rd_vlink("/lib/filters/avalue.o");

    /* Step through attribute values creating subdirs */
    cl = dir->links;
    while(cl) {
      attributes = pget_at(cl,argv[0]);
      for(ca = attributes;ca;ca = ca->next) {
        /* If not first link, then make copy */
        if(nd->links) sd = vlcopy(nd->links,0);

        /* Set name of new subdir and insert it */
        sd->name = stcopyr(ca->value,sd->name);
        if(vl_insert(sd,nd) == PSUCCESS) {
          /* If successful, then set filter arguments */
          /* Find last filter on current subdir       */
          for(avf=sd->filters;avf->next;avf=avf->next);
          sprintf(farg,"%s %s",ca->aname,ca->value);
          avf->args = stcopyr(farg,avf->args);
        }
      }
      atlfree(attributes);
      cl = cl->next;
    }
    /* Return the result in the original directory*/
    vdir_copy(nd,dir);
    return(dir);
  }
```

Figure 4: Simplified code for the distribute filter

Figure 4 shows code implementing the **distribute()** filter. The **distribute()** filter works by reading the value of the specified attribute for each file in the target directory, and creating a new link to the target directory for each distinct value. The name of the new link is the value of the attribute, and a filter is attached that selects only those files whose attribute matches that value.

Although users can write their own filters, most users can get by using the predefined ones. Among these are: **flatten()** take a directory hierarchy and make it appear like a single level name space, **match()** pass links matching a specified list of names, **matchhost()** pass links whose target is stored on the matched hosts, **distribute()** create subdirectories for each value of the specified attribute and distribute files among those directories according to that attribute, and **attribute()** pass only links for objects with attributes matching those specified.

Protection of objects in Prospero is based on the protection mechanisms associated with the underlying access method. The ability to resolve the name of an object does not grant permission to access the object. Access control lists may be associated with directories or individual links in directories. Such authorization attributes apply to the ability to resolve names in or to modify the directory. They do not apply to the referenced object itself[8].

[8]Though the attributes of an individual object may include the access control information for the under-

Storage site	Time to resolve a name in Prospero Number of components 1	2	3	4	5	Negotiate Access Method	Open Access Method	Time
Remote	38ms	76ms	115ms	153ms	191ms	32ms	NFS	125ms
Local	21ms	43ms	63ms	86ms	107ms	-	Local	27ms

Table 1: Approximate time to resolve a name and open a file

4.2 Performance

Table 1 shows the performance of the Prospero client on a DECstation 5000. The remote Prospero server is running on a second DECstation 5000 on the same Ethernet. The numbered columns represent the time required to resolve a name with the specified number of components. The second to last column is the time required to negotiate the access method and the final column is the time it takes to open the file. Since Prospero uses the existing access methods of the underlying system, the last column is also the time it takes to open the file without Prospero.

In compiling these figures, the optimization that resolves multiple components at the same time has been disabled. Thus, the time to resolve a name with consecutive components stored on the same server would be less; if all components are stored on the same server the time would be close to that in the first column.

The time required to open a file with a one component name using Prospero (name resolution + negotiation + open) is less than twice that required to open the file directly. This is very good when one considers that at least one extra pair of network messages is involved. Once a file is open, no additional overhead is incurred beyond the access times of the underlying file system. While the performance is quite good for a prototype, it is even better when considered in light of the real contribution of this work: that it enables users to better organize information.

5 Experience

Prospero has been available since December 1990. The prototype implementation allows users to construct virtual systems and to navigate through them. Programs linked with the Prospero compatibility library are able to specify file names relative to the active virtual system when opening files. In addition to the basic release, there are several standalone applications that rely on Prospero to retrieve directory information from indexing services. The prototype has been used to organize information on Internet sites world-wide and Prospero-based applications are used on more than 7,500 systems in 29 countries on six continents.

As distributed, a user's virtual system starts out with links to directories organizing information of various kinds in several ways. When a Prospero file name is mentioned in a mail or news message, the name space that was active when the message was sent appears in the header of the message. Recipients are thus able to properly resolve the name, as well as add links to the object from their own name spaces. This mechanism makes it easy for

lying method by which the object will be accessed.

a user to keep track of files of interest without having to retrieve the file right away. If the file moves and the storage site supports forwarding pointers (which will be the case if the file is moved using Prospero), the link to the file is updated when next referenced.

Users find information by moving from directory to directory in much the same manner as they would in a traditional file system. Figure 5 shows a sample session with Prospero. Users do not need to know where the information is physically stored. In fact, the files and directories shown in the example are scattered across the Internet. At any point, a user can access files in a virtual system as if they were stored on his or her local system.

In the example, the user connects to the root directory and lists it using the `ls` command. The result shows the categories of information included in the virtual system. The information includes online copies of papers (in the papers directory), archives of Internet and Usenet mailing lists (in the mailing-list and newsgroups directories), releases of software packages (in the releases directory), and the contents of prominent Internet archive sites (in the sites directory). Files of interest can appear under more than one directory. For example, a paper that is available from a prominent archive site might also be listed under the papers directory.

Next, the user connects to the papers directory, lists it, and finds the available papers further categorized as conference papers, journal papers, or technical reports. The technical report directory is broken down by organization, and by department within the organization. The journals directory is organized by the journal in which a paper appears, and the two journals that are shown are further organized by issue. Use of the `vls` command shows where a file or directory is physically stored, demonstrating the fact that the files are scattered across the Internet (IEEE TC/OS Newsletter on FTP.CSE.UCSC.EDU and Computer Communications Review on NNSC.NSF.NET.) Though not shown in the example, papers are also organized by author and subject in other directories from the same virtual system.

It is important to note that the example shows only part of the information available through Prospero, and that it shows a typical way that the information is organized. Individuals can organize their own virtual systems differently.

One of the most frequently used directories in Prospero is that representing the archie database, developed at McGill University [5]. That directory includes subdirectories organizing files according to the last components of their file names. For example, the subdirectory *prosp* contains references to the files available by Anonymous FTP whose names include the string "prosp". Among the matches would be files related to Prospero. The contents of each subdirectory are equivalent to what would result from running the Unix `find` command with appropriate arguments over all the major archive sites on the Internet (if it were even possible to do so). The subdirectories do not exist individually but are instead created when referenced by querying the archie database. The use of archie through Prospero has been so successful that the archie group has adopted Prospero as the preferred method for remote access to the archie database.

To provide the benefits of Prospero to users who have not installed it on their systems, Steve Cliffe of the Australian Academic and Research Network (AARNet) Archive Working Group has added Prospero support to one of their FTP servers. As well as making files available from the physical file system, the modified FTP server makes files available from a virtual file system. When a retrieval request is received, the FTP server locates the file

```
Script started on Wed Jan 29 21:02:50 1992
% cd /
% ls
afs             info            papers
databases       lib             projects
documents       mailing-lists   releases
guest           newsgroups      sites
% cd papers
% ls
authors           conferences       subjects
bibliographies    journals          technical-reports
% cd technical-reports
% ls
Berkeley    IAState       OregonSt     UCalgary   UWashington
BostonU     MIT           Purdue       UColorado  Virginia
Chorus      NYU           Rochester    UFlorida   WashingtonU
Columbia    NatInstHealth Toronto      UKentucky
Digital     OregonGrad    UCSantaCruz  UMichigan
% ls UCSantaCruz
crl
% ls UCSantaCruz/crl
ABSTRACTS.1988-89           ucsc-crl-91-01.ps.Z
ABSTRACTS.1990              ucsc-crl-91-02.part1.ps.Z
ABSTRACTS.1991              ucsc-crl-91-02.part2.ps.Z
ABSTRACTS.1992              ucsc-crl-91-02.ps.Z
INDEX                       ucsc-crl-91-03.ps.Z
ucsc-crl-88-28.ps.Z         ucsc-crl-91-06.ps.Z
...
% ls UWashington
cs   cse
%
% ls UWashington/cs
1991           INDEX          PRE-1991
1992           OVERALL-INDEX  README
% cd /papers
% ls
authors           conferences       subjects
bibliographies    journals          technical-reports
% ls journals
acm-sigcomm-ccr  ieee-tcos-nl
% ls journals/ieee-tcos-nl
app-form.ps.Z  v5n1           v5n3
cfp            v5n2           v5n4
% ls journals/acm-sigcomm-ccr
application.ps  jan89           jul90           sigcomm90-reg.ps
apr89           jan90           oct88
apr90           jan91           oct89
apr91           jul89           sigcomm90-prog.ps
% vls journals
   acm-sigcomm-ccr       NNSC.NSF.NET     /usr/ftp/CCR
   ieee-tcos-nl          FTP.CSE.UCSC.EDU /home/ftp/pub/tcos
%
script done on Wed Jan 29 21:06:53 1992
```

Figure 5: Sample session

using Prospero and checks to see if a copy of the file is available locally. Using Prospero to check the last modified time of the authoritative copy, the FTP server checks that the local copy is current. If a current copy does not exist locally, the server retrieves and caches a copy of the file. The local copy is then returned to the client.

6 Related Work

Allowing users to construct a view of a system by selecting components that are available on the network is a goal that is shared by Plan 9. One of the key differences is that Plan 9 addresses the problems of combining the components, not of finding them. The system components in Plan 9 have global names. Plan 9 does not address the problem of how users identify the components that they want to include in their system view. Prospero is concerned primarily with the mechanisms needed to organize and identify the components of interest and relies on system provided access methods to actually use them.

The functionality of filters in Prospero is similar to the domain-switching portal mechanism found in the Universal Directory Service [9]. A portal is a call to a separate name server that may have a non-standard implementation, enabling it to resolve names in a manner different than that in a standard name server. A portal is implemented as a separate server, while a filter is executed by the name resolver. Though the result of resolving a name through a portal is a function of the remaining components in the name, the result is not affected by the point at which the portal is attached. This means that a new portal (and hence new name server) must be run for each point of attachment. The portal mechanism is closer to that used to integrate Prospero and archie than it is to most filters.

Attribute-based naming, supported by Profile [14] and the Semantic File System[6], provides an alternative mechanism for finding information of interest. In attribute-based naming a database is maintained of object attributes, and the user specifies the known attributes of an object instead of its name. In the Semantic File System, the result is a directory listing those objects matching the specified attributes. In Profile, if enough attributes have been specified to uniquely identify the object, the result is a reference to the object itself.

For attribute-based naming to scale, directory information must be distributed across multiple servers. Without a way to direct a query to the right server, queries must be sent to all servers, an operation that doesn't scale. Profile restricts the set of servers that are queried and relies on cross-references to direct queries to servers that were not included in the original set, but doing so negates one of the advantages of attribute based systems; the necessary cross-references must be in place before a query is made.

When used together, attribute-based naming and Prospero could be very powerful. The databases maintained by such systems could be accessed through filters that could perform any desired pre- or post-processing. Other features of the Virtual System Model could be used to impose a structure that directs queries to the appropriate servers. Such a combination of attribute- and link-based naming is similar to recent work on multi-structured naming [22].

In an alternative approach to finding objects in large systems, Schwartz proposes the use of resource discovery agents [21] that accept queries from users and use the information provided by the user to find objects in which the user is interested. In Schwartz's design, the information needed to direct a query to the appropriate agent evolves over time. A query

is directed to the nearest agent, and agents learn how to direct queries based on the results of previous queries. The problem with this approach is that it gives the agents too much of the responsibility for building the resource discovery graph. However, a combined approach where agents make use of the organization imposed by individuals (e.g., as encoded in the Prospero naming network) might yield better results.

7 The Future of Prospero

Prospero is an evolving system. We continue to collaborate with other groups to extend it. We are in the process of adding support for the retrieval of documents maintained by the Wide Area Information Service (WAIS) [8]. We plan to add filters that access directory information maintained by by semantic file systems [6] and distributed indices [4] when those systems are available.

We plan to implement a new application interface for Prospero in order to allow use by existing applications without relinking. This will be accomplished by adding Prospero support to an NFS server [20], the same approach taken by semantic file systems [6] and Alex [2]. We hope to benefit from changes already made in those systems.

The Prospero protocol provides a lightweight protocol for querying directories and obtaining file attributes. We encourage its use as a base upon which other systems can be built. Archie and AARNet are two examples. It is being considered for use by Alex [2] to improve the performance queries to directories on hosts that run Prospero servers.

8 Conclusions

This paper discussed several problems that arise in the organization of a global file system. It demonstrated the importance of customization and presented two mechanisms, the filter and the union link, that allow views of the global name space to be defined in terms of other views. The lack of name transparency across customized name spaces has the potential to cause confusion, but this problem is addressed by supporting closure.

The prototype file system and directory service described in this paper is used from more than 7,500 systems worldwide. The use of the prototype to solve real problems was discussed; its acceptance demonstrates the benefits of the organizational flexibility provided by the Virtual System Model.

Availability

To find out more about Prospero, or for directions on retrieving the latest distribution, please send a message to info-prospero@isi.edu.

[11] B. Clifford Neuman. Workstations and the Virtual System Model. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 91–95, September 1989. Also appears in the *Newsletter of the IEEE Technical Committee on Operating Systems*, Volume 3, Number 3, Fall 1989.

[12] B. Clifford Neuman. *The Virtual System Model: A Scalable Approach to Organizing Large Systems.* PhD thesis, University of Washington, 1992. Department of Computer Science and Engineering (in preparation).

[13] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–35, February 1988.

[14] Larry L. Peterson. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[15] Larry L. Peterson and Herman C. Rao. Accessing files in an internet: The Jade file system. Technical Report TR 90-30, University of Arizona, 1990.

[16] Jon B. Postel. User datagram protocol. DARPA Internet RFC 768, August 1980.

[17] Jon B. Postel and J. K. Reynolds. File transfer protocol. DARPA Internet RFC 959, October 1985.

[18] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9: A distributed system. In *Proceedings of Spring 1991 EurOpen*, May 1991.

[19] Jerome H. Saltzer. *Operating Systems: an advanced course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3, pages 99–208. Springer-Verlag, 1978.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.

[21] M. F. Schwartz. Resource discovery and related research at the University of Colorado. Technical Report CU-CS-508-91, Department of Computer Science University of Colorado, Boulder, January 1991.

[22] Stuart Sechrest and Michael McClennen. Blending hierarchical and attribute-based file naming. In *Proceedings of the 12th International Conference on Distributed Computer Systems*, June 1992.

[23] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experience with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):47–63, December 1990.

[24] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The Locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

Acknowledgments

Ed Lazowska provided valuable guidance throughout this work. Discussions with John Zahorjan, Hank Levy, and Alfred Spector helped me refine the ideas that ultimately led to the development of Prospero. Celeste Anderson, Ben Britt, Vincent Cate, Steve Cliffe, Peter Danzig, Peter Deutsch, Alan Emtage, Deborah Estrin, and Dennis Hollingworth provided comments on earlier drafts of this paper.

References

[1] Luis-Felipe Cabrera and Jim Wyllie. QuickSilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 23–27, March 1988. Also IBM Research Report RJ 5578, April 1987.

[2] Vincent Cate. Alex: A global file system. In *Proceedings of the Workshop on File Systems*, May 1992.

[3] Douglas Comer, Ralph E. Droms, and Thomas P. Murtagh. An experimental implementation of the Tilde naming system. *Computing Systems*, 4(3):487–515, Fall 1990.

[4] Peter B. Danzig, Jongsuk Ahn, John Noll, and Katia Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, October 1991.

[5] Alan Emtage and Peter Deutsch. archie: An electronic directory service for the Internet. In *Proceedings of the Winter 1992 Usenix Conference*, pages 93–110, January 1992.

[6] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.

[7] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[8] Brewster Kahle and Art Medlar. An information system for corporate users: Wide area information systems. Technical Report TMC-199, Thinking Machines Corporation, April 1991.

[9] Keith A. Lantz, Judy L. Edighoffer, and Bruce L. Hitson. Towards a universal directory service. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.

[10] B. Clifford Neuman. The need for closure in large distributed systems. *Operating Systems Review*, 23(4):28–30, October 1989.

A Comparison of the Vnode and Sprite File System Architectures

Brent Welch
welch@parc.xerox.com
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

This paper compares the vnode architecture found in SunOS with the internal file system interfaces used in the Sprite distributed file system implementation. The emphasis of the comparison is on generalized support for remote access to file system resources, which include peripheral devices and IPC communication channels as well as regular files. A strong separation of the internal naming and I/O interfaces is exploited in Sprite to easily provide remote access to all of these resources. In contrast, the vnode interface mixes naming and I/O operations in a single interface, making generalized remote access more awkward.

1 Introduction

Kernel support for remote file systems is centered around internal file system interfaces that hide the details of accessing file systems whether they are local or remote. Perhaps the most widely known of these interfaces is the SunOS vnode interface [Sandberg85][NFS85], although Ultrix has a similar gnode interface[Rodrigeuz86], and ATT UNIX has a remote inode interface[Rifkin86]. These interfaces are all evolutions of the original UNIX file system implementation in order to support remote file systems. The basic approach was to abstract the file and directory access procedures so there could be different implementations corresponding to different sorts of file systems. Examples include the original ATT UNIX local file system, the BSD fast file system format, and remote file systems such as NFS, AFS, and RFS. As well as introducing a generic interface, the original inode data structure was replaced with a more opaque object descriptor, which is called a *vnode* in SunOS, that represents objects implemented by different file systems.

The main property of all these designs is that they are oriented mainly towards file access, and this bias leads to a fundamental problem: naming and I/O operations are lumped together into the same internal file system interface. For example, there are operations on vnodes that locate their name in a directory, as well as operations that transfer data to the object represented by that name. By grouping these two different classes of operations into the same interface, the option of implementing pathnames by dedicated name servers is virtually precluded. Instead, the server for an object must implement both naming and I/O operations on its objects. For example, RFS provides remote device access, but you have to mount the /dev directory of the remote machine in order to access its devices. While this is reasonable for files, it is less convenient for peripheral devices, especially in an environment of diskless workstations.

In contrast to the designs that evolved from a UNIX implementation, the Sprite file system

architecture provides generalized remote access to different kinds of file system resources that include peripheral devices and IPC channels as well as files and directories. The goal of the Sprite architecture is to use the distributed file system as a name server for objects other than files, and to do this is such a way that does not require a local file system on each workstation. The key to achieving this is to have two distinct internal file system interfaces, one for pathname operations and one for I/O operations. This makes it possible to have a pathname implemented by Server A that corresponds to a peripheral device on Client X, or an IPC channel to a process on Client Y.* The main point is that by properly separating the naming and I/O interfaces it is quite feasible to reuse the file server's directory structures as a global name space for any object accessed via the UNIX open-close-read-write interface.

The remainder of the paper discusses the vnode and Sprite architectures in more detail. Notable features of the Sprite architecture include a prefix caching system that replaces the UNIX mount mechanism, support for process migration and crash recovery, and an object-oriented design that has two base classes: pathnames and I/O objects.

2 The Vnode Architecture

The vnode architecture has two internal interfaces, the vnode and vfs interfaces. The vfs interface is concerned with the mount mechanism, while the vnode interface is concerned with access to objects within a file system.

The operations in the vnode interface are listed in Table 1. This is the interface as defined for SunOS 4.1.1. The details of each operation are not crucial to the arguments presented here, although some operations will be discussed in more detail below. The main thing to note that there is a rough classification of the operations into two sets. The first set of operations deal with pathnames: vn_access, vn_lookup, vn_create, vn_remove, vn_link, vn_rename, vn_mkdir, vn_rmdir, vn_readdir, vn_symlink, and vn_readlink. The other set of operations apply to the underlying object being named by a pathname (e.g., a file or a device). These operations include vn_open, vn_close, vn_rdwr, vn_ioctl, vn_select, vn_getattr, vn_setattr, vn_fsync, vn_lockctl, vn_getpage, vn_putpage, and vn_map. There is also a set of routines that deal more with the management of the vnode data structures themselves: vn_inactive, vn_fid, vn_dump, vn_cmp, vn_realvp, vn_cntl.

The vfs interface operations are given in Table 2. This interface is primarily concerned with the process of mounting a file system into the global directory hierarchy. The mount mechanism assembles self-contained directory structures on different disks (unfortunately called "file systems") into a single hierarchy. Each file system has a root directory. One file system is the distinguished root of the overall hierarchy. The vfs_mount operation is used to mount the root directory of other file systems onto an existing directory. Ordinarily file systems are mounted onto top-level directories of the root file system (e.g., /usr or /vol1), but it is legal to mount file systems on any directory, even one in a mounted file system (e.g., if disk 1 is mounted on /a, then disk 2 could be mounted on /a/b, or even /a/b/c).

---

*. Generally, the terms "server" and "client" refer to hosts and their operating system kernel.

Table 1. Vnode Operations <sys/vnode.h>

Operation	Description
vn_open	Initialize an object for future I/O operations.
vn_close	Tear down the state of the I/O stream to the object.
vn_rdwr	Read or write data to the object.
vn_ioctl	Perform an object-specific operation.
vn_select	Poll an object for I/O readiness.
vn_getattr	Get the attributes of the object.
vn_setattr	Change the attributes of the object.
vn_access	Check access permissions on the object.
vn_lookup	Look for a name in a directory.
vn_create	Create a directory entry that references an object.
vn_remove	Remove a directory entry for an object.
vn_link	Make another directory entry for an existing object.
vn_rename	Change the directory name for an object.
vn_mkdir	Create a directory.
vn_rmdir	Remove a directory.
vn_readdir	Read the contents of a directory.
vn_symlink	Create a symbolic link.
vn_readlink	Return the contents of a symbolic link.
vn_fsync	Force modified object data to disk.
vn_inactive	Mark a vnode descriptor as unused so it can be uncached.
vn_lockctl	Lock or unlock an object for user-level synchronization.
vn_fid	Return the handle, or file ID, associated with the object.
vn_getpage	Read a page from the object.
vn_putpage	Write a page to an object.
vn_map	Map an object into user memory.
vn_dump	Dump information about the object for debugging.
vn_cmp	Compare vnodes to see if they refer to the same object.
vn_realvp	Map to the real object descriptor.
vn_cntl	Query the capabilities of the object's supporting file system.

Table 2. Vfs Operations <sys/vfs.h>

Operation	Description
vfs_mount	Mount a file system onto a directory.
vfs_unmount	Unmount a file system.
vfs_root	Return the root vnode descriptor for a file system.
vfs_statfs	Get file system statistics.
vfs_sync	Flush modified file system buffers to safe storage.
vfs_vget	Map from a file ID to a vnode data structure.
vfs_mountroot	Special mount of the root file system.
vfs_swapvp	Return a vnode for a swap area.

There are two problems with the vnode interface design that limit its ability to provide generalized remote access and to share the file system uniformly throughout a network. One problem is that each host defines its own mount table. This stems from UNIX's past as a single-host, timesharing system. The other problem is that the vnode interface includes both naming and I/O operations. This stems from the file-oriented bias of the vnode architecture. Both problems are also related to the way the pathname resolution algorithm was extended to a distributed environment.

In a stand-alone system, pathname resolution involves processing a pathname one compo-

nent at a time by looking for that component in the current directory of the search. Each directory is checked to see if it is a mount point for another file system. If it is, the current directory of the search shifts to the root of the mounted file system. Symbolic links are also expanded during this process. Thus, this basic algorithm is a component-by-component traversal of a pathname, with indirections at mount points used to glue together file systems on different disks.

In a distributed system there are a number of places to split the pathname resolution algorithm between clients and servers. LOCUS put the split at the block access level so that remote pathnames were resolved by reading remote directory blocks over the network [Walker83]. The vnode architecture puts the split at the component access level. The vn_lookup operation takes a directory handle and pathname component as arguments and returns the handle on the named file, if it is found. Similarly, the contents of symbolic links are retrieved through the vn_readlink call. Sprite and RFS put the split at the pathname access level so that component-by-component iteration happens on the server. The appeal of a lower-level split is that there is less effect on existing higher-level code. The advantage of a higher-level split is that more functionality can be moved to the server and the number of client-server interactions can be reduced. Note that the vnode architecture tries to optimize by keeping a cache of recent name translations. However, an NFS server cannot guarantee the consistency of this cache, so each entry for remote files remains valid for only a few seconds on the client. In general, a higher-level interface hides more details from the client and gives more flexibility to the server.

The vnode design also requires that each client maintain its own mount table so that its lookup procedure can do the indirection at mount points. As a system gets larger, the job of keeping all the clients' mount tables consistent becomes a problem. SunOS introduced an automounter mechanism to work around the problem of mount table consistency. The basic idea of an automounter is that a user process maintains a map from mount points to file systems. The automounter maps are kept in a distributed database service, NIS. The automounter process postpones the mounting of file systems by mounting itself onto top level mount points. When an access is made to a "file system" mounted on the automounter process, the automounter does the real mount and returns in such a way that the operation is retried. The point of delaying the kernel-level mounts is to increase availability. First, using a file system served by a crashed server can hang a process in many NFS implementations. Second, if a file system is replicated, then postponing the mount increases the chance that the file system will be mounted from a working server. Thus, by retaining the mount mechanism, a new automounting mechanism has to be introduced to keep mount tables consistent across clients.

So far it seems like the problems in the vnode interface can be fixed up. However, perhaps the biggest problem is the mixing of naming and I/O operations in the vnode interface. The mixture is easy to fall into from the viewpoint of a simple file system of regular files and directories. Directories are just special cases of files that name other files. Both objects are resident on the same disk, so the same host can be the server for both, and the same internal data structure (vnode) can be used for both. But consider a peripheral device. First, there are often many different names that map to the same device. With tape drives, for example, different names imply different tape densities and whether or not the tape is rewound when it is closed. This creates the following problem. The result of pathname resolution is a

vnode that corresponds to the special file that names the device. In order to perform I/O on the device, another data structure, the *snode* that corresponds one-to-one with the device, is needed to serialize operations and maintain state about the device. Under the covers, the special device file system implementation must maintain snodes and arrange for all the vnodes associated with device names to have a pointer to the real object descriptor, the snode. The vn_realvp operation in the vnode interface is used to map from vnodes to snodes.

Again, there seems to be a fix. But what about remote device access? The mixture of naming and I/O operations implies that if you want to provide access to a peripheral device then you also need to export its name. This distributes the responsibility for naming among all servers. In UNIX terms, it means that you have to mount the /dev directory of a remote host into your file system in order to access its peripheral devices. Even a diskless workstation needs its own file system that has a /dev directory with names for its peripheral devices. This forces all clients to maintain a file system, which increases administrative overhead. It can also increase cost and noise if it means adding local disks.

Forcing each host to have its own file system may not seem like such a bad thing. After all, a workstation might be able to function with its own file system if the file servers are unavailable. However, in practice most workstations are configured so that they depend heavily on file servers, in spite of local storage. Only the bare minimum is on the local file systems, and most important files, including system programs, are on remote servers. This approach is taken to reduce administration effort. In such an environment, why is it necessary to have a local file system at all?

What about extending the vnode architecture to a multicomputer where the hosts are even more tightly bound than in workstation environments? Clearly it should not be necessary to maintain 1024 nearly identical private file systems just so each node can mount together file systems. On the other hand, if file system operations are forwarded to distinguished file server nodes, then the lookup traffic in the root directory becomes a bottleneck. What is needed is a system that efficiently partitions the name space among different servers, and then separates naming and I/O so that inter-node device access does not involve file server nodes.

In summary, the vnode and related architectures result in a world of many remote file systems that happen to be accessible. Instead, I prefer a world with a single file system view that transparently incorporates the resources available on all machines without highlighting machine boundaries.

3 The Sprite Architecture

The Sprite file system architecture presented here is a second-generation design that was implemented after initial experience with Sprite, which we began building in 1985. The current design has been operational since 1989. The goal of the architecture is to generalize the remote access capabilities that support remote file access to also accommodate remote device and remote IPC access. In Sprite, remote access also involves state management to support process migration and failure recovery, not just I/O.The architecture starts by cleanly separating naming and I/O operations so that file servers can easily act as name servers for devices and inter-process communication (IPC) channels. It also eliminates the need for private client file systems, and it makes a different cut between client and server

in pathname operations that is more efficient than the component-level split in the vnode interface.

As with the vnode design, the Sprite design uses a generalized *object descriptor*, which is a main-memory data structure maintained by the kernel, not a disk-resident representation of an object. A basic object descriptor has a type, uid, server ID (a Sprite Host ID), a reference count, and a lock bit. Objects are specified internally by a tuple of <type, serverID, uid>. This base data structure is subclassed* for the implementations of various object types. The object-oriented approach allows clean separation of different object implementations, as well as sharing between similar objects like remote devices and remote pipes. A diagram of the file system architecture is given in Figure 1.

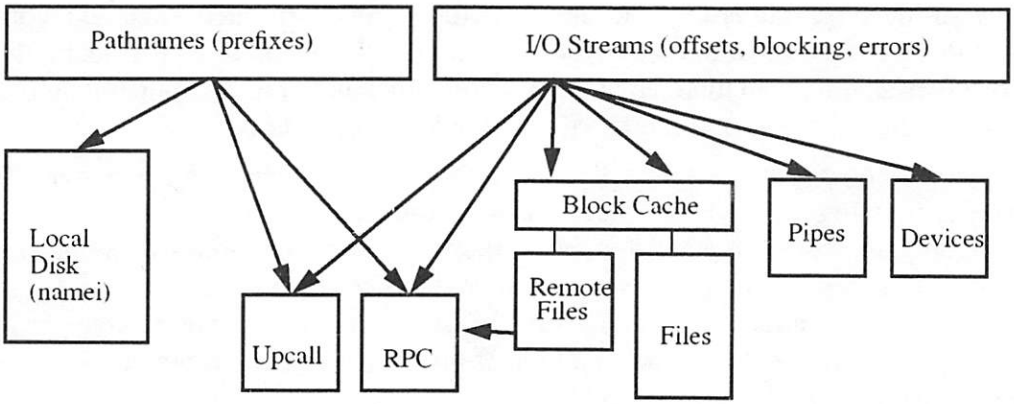


Figure 1. An overview of the Sprite file system architecture. The two primary interfaces involve pathnames and I/O streams. The Upcall module forwards operations to user-level processes. RPC forwards operations across the network to other Sprite kernels.

The pathname interface illustrates the three basic cases handled by the Sprite kernel. The server for a pathname may be the local kernel, in which case the file system implementation is accessed by an ordinary procedure call within the Sprite kernel. The server may be remote, in which case a kernel-to-kernel RPC protocol is used to pass the pathname to the server [Welch 86a]. Finally, the server may be a user-level process, in which case an upcall mechanism, which was described in [Welch 88], is used to pass the operation up to a user-level *pseudo-device* or *pseudo-file-system* server process. Thus there are three orthogonal cases that are supported by a Sprite kernel, a local, kernel-resident module, a remote module, and a user-level module.

3.1 Separating Naming and I/O

Consider the UNIX open system call that maps from a pathname to an I/O stream. In Sprite, this is broken into two operations, NameOpen and IoOopen, that involve both of the internal file system interfaces. The NameOpen procedure returns attributes of the named object. The IoOpen procedure uses these attributes to create an open I/O stream. The NameOpen and IoOpen procedures may be implemented by different servers, and this is achieved cleanly by branching through the object-oriented naming and I/O interfaces.

*. The various kinds of object descriptors would be the result of subclassing if Sprite were written in C++. However, all the object-oriented features described here were hand crafted in C.

The clean separation of naming and I/O means that objects like devices can rely on a file server to implement the naming interface on their behalf. In Sprite, special files are used to represent devices and pseudo-devices in the name space. Pseudo-devices can be considered a kind of IPC channel [welch 88]. Furthermore, a Sprite file server can have special files that represent devices and pseudo-devices on any machine in the network. Contrast this with NFS, which doesn't support remote device access, or even RFS, which only supports accesses to devices on the file server. Those systems are limited by the vnode (or equivalent) interface that lumps naming and I/O together.

The Sprite naming and I/O interfaces are described in Tables 3 and 4.

Table 3. Sprite Naming Interface <fs/fsNameOps.h>

Operation	Description
NameOpen	Map from a pathname to attributes of an object, and prepare for I/O.
GetAttributes	Map from a pathname to attributes of an object.
SetAttributes	Update the attributes of an object.
MakeDevice	Create a special file that represents a device.
MakeDirectory	Create a directory.
Remove	Remove a named object.
RemoveDirectory	Remove an empty directory.
HardLink	Create another name for an existing object.
Rename	Change the pathname of an existing object.
SymLink	Create a symbolic link.

Table 4. Sprite I/O Interface <fsio/fsio.h>

Operation	Description
IoOpen	Complete the preparation for I/O to an object.
Read	Read data from the object.
Write	Write data to the object.
PageRead	Read a page from a swap file.
PageWrite	Write a page to a swap file.
BlockCopy	Copy a block of a swap file. Used during process creation.
IoControl	Perform an object-specific operation.
Select	Poll an object for readability, writability, or an exceptional condition.
IoGetAttributes	Fetch attributes that are maintained at the object.
IoSetAttributes	Change attributes that are maintained at the object.
ClientVerify	Verify a remote client's request and map to local object descriptor.
Release	Release references after an I/O stream migrates away.
MigEnd	Acquire new references as an I/O stream migrates to a host.
SrvMigrate	Update state on the I/O server to reflect an I/O stream migration.
Reopen	Recover state after a server crash.
Scavenge	Garbage collect object descriptors.
ClientKill	Clean up state associated with a client.
Close	Clean up state associated with I/O to an object.

The I/O interface has more cases to handle the different kinds of objects implemented by kernel-resident modules. The cases are specified in detail by the types of the object descriptors listed in Table 5. There are corresponding local and remote cases for various kinds of objects like files, pipes, devices, and pseudo-devices. Most of the remote cases share their implementations of the Read, Write, Ioctl, and Select operations, which are just RPC stubs. The state-related operations differ, although often not greatly. The remote file implementa-

tion is optimized to use the local cache, although in the case of a cache miss it uses the same RPC routines as the other cases. In the case of a remote operation, the operation passes through the pathname or I/O interfaces again on the server, so all object implementations are accessible remotely. The importance of this approach is that it extends the general features implemented in upper levels of the kernel to local, remote, and user-provided objects. Notable, high-level features include the name space, error recovery, and blocking I/O. Thus, the focus of the file system architecture has relatively little to do with actual disk management. The focus is on extending the high-level abstractions of pathnames and I/O streams to the network environment.

Table 5. Sprite Object Descriptor Types <fsio/fsio.h>

LCL_FILE	A file, directory, or symbolic link stored locally.
RMT_FILE	For a remote file, directory, or link.
LCL_DEVICE	A device on this host.
RMT_DEVICE	A device on a remote host.
LCL_PIPE	An anonymous pipe buffered on this host.
RMT_PIPE	An anonymous pipe buffered on a remote host.
PDEV_CONTROL	Used by a pseudo-device server to listen for connection requests.
SERVER	The upcall channel to a pseudo-device server.
LCL_PSEUDO	Client's handle on a local upcall channel.
RMT_PSEUDO	Client's handle on a remote upcall channel.
PFS_CONTROL	Used by a pseudo-file-system server to listen for connections.
PFS_NAMING	Upcall channel used for naming operations in a pseudo-file-system.
LCL_PFS	Client's handle on local upcall channel to pfs server.
RMT_PFS	Client's handle on remote upcall channel to pfs server.
RMT_CONTROL	Used during get/set I/O attrs if pseudo-device server is remote.
PASSING	Used to pass existing I/O streams from a pdev server to a client.

3.2 Integrating User-Level Servers

Pseudo-devices are user-level processes that implement the I/O interface by way of an upcall mechanism [welch88]. They are used to implement the X server, terminal emulators, and a TCP/IP server. Unlike UNIX ptys that use pairs of special device files, there is a single pathname that represents a pseudo-device in Sprite. A pseudo-file-system is a server* that implements the pathname interface, and this is used to implement a gateway to NFS file systems.

The upcall mechanism is used to forward system calls on these objects up to their user-level server. A pseudo-device server is represented by a special file, in a similar way that special device files are used in UNIX. The server opens the file and can listen for connections by clients. When a client opens the pseudo-device file, a new upcall channel is created for communication, and the server process gets a new open file descriptor to represent it. Pseudo-file-systems are similar, except that the server is registered in the prefix table mechanism, which is described below. An upcall channel is used to forward pathname operations to the server process, and open calls result in new upcall channels corresponding to the client's open I/O stream.

The upcall implementation has a number of object descriptor types, although there are really only 4 different kinds of object descriptors used. Extra types were introduced to han-

*. In the context of pseudo-devices, "clients" and "servers" are user-level processes.

dle special cases, sort of a poor man's subclassing mechanism. For each pseudo-device there is one control descriptor that keeps state about where the user-level server process is executing. The control descriptor is also used by the server to listen for client connections. 2 descriptors are used to represent the client and server sides of an upcall channel. The passing descriptor is used to return an open I/O stream in response to a pseudo-file-system open request, which is an alternative to creating an upcall channel. This could be used to open files or devices via a pseudo-file-system, although this is still not fully implemented and debugged.

The Plan-9 system also integrates user-level server processes into its name space. It uses the mount system call to attach a full-duplex pipe to a name. All client operations, both naming and I/O, from all clients, are passed through this channel along with a unique tag so the server can manage requests. The main difference from Sprite pseudo-device and pseudo-file-systems is that the Sprite kernel maintains separate channels corresponding to each client open system call, and in the case of pseudo-devices it implements the naming interface on behalf of the server.

3.3 Handling Special Files

A final touch is required to implement support for special files cleanly on the file server. After a file server finds a file in one of its domains, it needs to take type-specific action to complete the servicing of a NameOpen request. This special action is what distinguishes a NameOpen operation from a GetAttributes operation. Another, single-procedure interface called the SrvOpen interface is used to abstract these type-specific actions. The implementation of SrvOpen is selected based on a type field in the disk-resident descriptor for a file (the disk inode in UNIX terms). Currently there are three different implementations of the SrvOpen procedure.

The first implementation is used for regular files, directories, and symbolic links. This procedure invokes the Sprite cache consistency algorithm [Nelson 88] and sets up enough state about the remote client so that the IoOpen procedure does not need to contact the file server a second time. This is an important optimization for the common case of file access.

A second implementation is used for device files. In this case the procedure just has to extract the relevant attributes from the disk-resident descriptor that are needed by the remote client. Sprite device files have a ServerID attribute in addition to the standard UNIX major and minor device numbers. The ServerID identifies what host controls the device (i.e., the I/O server). A simple trick is played here to allow sharing of the /dev directory. A special value of the ServerID is mapped to the host ID of the client making the NameOpen RPC, thus mapping the special device file to the instance of the device on that client. This trick makes it easy to share a single /dev directory that is shared among all the hosts in the Sprite network. Most entries in /dev are these generic device files that map to the local instance. Some entries have specific ServerID attributes so they map to devices on particular clients. Finally, note that this arrangement results in a full many-to-many mapping from pathnames to objects that really requires a clean separation of the naming and I/O interfaces.

The third implementation of SrvOpen is used for pseudo-devices. The file server maintains state about which host the pseudo-device server process is executing on, and it updates and verifies this state when a pseudo-device is opened. Server processes supply an extra flag on

the open call to distinguish themselves, and after that one or more clients can open the pseudo-device and get a connection to the server.

The vnode architecture has a "cloning" mechanism that is used to achieve a similar effect of having distinct NameOpen and IoOpen procedures. The vnode returned from the vn_open call can be cloned to get better handle on the underlying object. For example, this is when the snode that corresponds to a device is located and linked back to the vnode. However, this is a client-side operation that is invoked after the server does the lookup, and it is oriented towards the special case of special device files.

4 The Sprite Prefix Table Mechanism

Sprite prefix tables were originally described in [Welch 86b], but the following description is included for completeness.

Sprite uses a prefix table mechanism to implement a uniformly shared, hierarchical name space. Each Sprite kernel keeps a cache of pathname prefixes. The prefixes define the way server domains are coalesced into a single hierarchy, and their use completely replaces the UNIX mount mechanism. The Sprite naming protocol ensures that servers export their domains consistently so that all hosts, and therefore all processes, see exactly the same name space. Users, administrators, and developers enjoy the simplicity of a single, shared name space. The fully shared file system supports cross-compilation and easy maintenance for all architectures from any workstation.

In contrast, the V-system [Cheriton87] and Mach 3.0 use a prefix cache that is maintained on a per-process basis by library routines. While this is advertised as a feature that allows custom name spaces, I believe this is a case where generality is not what you want. I think that the real reason V and Mach provide per-process prefix caches is because they are implemented in the run-time library instead of the kernel. Plan-9 [Pike90] relies heavily on per-process name spaces created by mounting various servers into the name space. Sprite also mounts services into the name space, even user-level processes as described below, but again, the name space is global and shared by all processes.

The basic idea of prefix caching is simple. On a client, the longest matching prefix selects the *domain* for the pathname. An object ID is registered with the prefix that identifies the server for the domain, the domain's type (e.g., local, remote, or user-level), and a uid that identifies the domain to the server. The server is sent the part of the pathname after the prefix, along with the object ID. Relative pathnames bypass the prefix match and are sent to the server of the current working directory, along with the object ID for the current directory. Note that the interface to the server is the same in both cases. In summary, clients match prefixes to choose servers. Servers process pathnames relative to one of their domains. When servers completely resolve a pathname, they perform the requested pathname operation (NameOpen, Remove, Rename, MakeDirectory, etc.).

4.1 Pathnames that Involve Multiple Servers

The main complication with prefix matching is that a pathname can wander through many different server domains, so the initial prefix match may not lead to the right server. The solution to this problem is to return control to the client when a pathname leaves a server's domain. This gives the client an opportunity to cache information about the next server, including a prefix that may optimize future pathname operations.

A pathname exits a domain when it encounters a symbolic link to an absolute pathname, or if it specifies ".." in the domain's root directory. In these cases, the server returns a new pathname to the client. The returned pathname is either the new absolute pathname resulting from the symbolic link expansion, or a pathname that begins with "..". The client combines the latter form with the prefix used to select the server in order to get a new absolute pathname. The ".." and the last component of the prefix are removed so the pathname will match on a different prefix.

Mount points are handled by placing a special symbolic link at the mount point called a *remote link*. Mount points can occur in any directory, so it is possible to nest server domains arbitrarily. The contents of a remote link is the prefix, or absolute pathname, of the mount point. The link has a different file type than ordinary symbolic links so that the server knows when it hits a mount point. The server expands the remote link and returns the new pathname to the client. In addition, the server indicates of how much of the returned pathname is the prefix of the mount point so that the client can add the prefix to its cache. Note that there is nothing in the remote link but its own name, so some other mechanism is used to locate the server for that domain. Currently, Sprite uses broadcast to locate the server. After locating the server and obtaining the object ID for the root directory of the domain, the client reiterates the lookup procedure. On the next iteration the returned pathname will match on the new prefix, and the lookup will be directed to the next server.

The iteration over the prefix table is contained within two routines on the client, one for single pathname operations and one for two pathname operations. Higher-level procedures such as Fs_Open or Fs_Rename (these implement the open and rename system calls) package up their non-pathname parameters into a structure and call the appropriate prefix table routine. The arguments to the call are the pathname(s), an operation identifier (e.g., for NameOpen or Rename), the structure that collects the remaining miscellaneous arguments, and a structure for the return values of the operation. The prefix table routines do the prefix match, branch through an operation table depending on the server's type and the requested operation, and handle the case where pathnames are returned from the server by reiterating the procedure. In the call to the server, the pathname arguments are replaced by an object ID, relative pathname pair.

Bootstrapping is achieved by broadcasting for the server of "/". Initially all lookups are directed to the root server. Remote links will cause prefixes to be returned to the client. As a client's prefix cache fills up, the prefix matches usually direct operations to the correct server, bypassing the servers for higher levels of the directory hierarchy.

Measurements of the Berkeley Sprite network revealed that about 17% of pathnames suffered redirections back from a server, but less than 1% of the pathnames were redirected because they hit a remote link [welch90]. Instead, ordinary symbolic links between domains caused the bulk of the redirections. It should be possible to extend the prefix caches to cache the results of absolute symbolic link expansion so that these redirections are eliminated, but this has not been implemented.

4.2 Operations on Two Pathnames

The Rename and HardLink operations involve two pathnames, and they are constrained to operate on pathnames in the same domain. The problem is that the two pathnames may or may not start out in the same domain, and they may or may not end up in the same domain.

The required iteration over the prefix matches is described below. Note that in the best case only a single server operation is required. By making Rename, and HardLink, into a single server operation, the server can easily implement these atomically without having to maintain state between multiple client operations.

The client matches both pathnames against the prefix table and obtains object IDs and relative pathnames for both. It sends these to the server identified by the object ID for the first pathname. The server traverses the first pathname. If the pathname leaves its domain, it returns a new pathname to the client as described above. In this case, the client reiterates the prefix matches and retries the operation. If the first pathname terminates in the server's domain, then it proceeds to traverse the second pathname. If the second pathname exits the domain, then the server returns EXDEV to indicate a potential conflict. At this point the client has to verify the conflict because it is still possible for the second pathname to end up back in the same domain as the first one. To verify the conflict, the client does a GetAttributes of the parent directory of the second pathanme. The parent is checked to avoid file-not-found errors. If the GetAttributes results in a returned pathanme, then the Rename or Hardlink operation is retried. Eventually the iteration terminates with a successful server operation or a verification that the two pathnames are in different domains.

4.3 Pros and Cons

There are a number of good properties of the Sprite prefix mechanism, and one limitation. First, clients are simplified. They do not iterate through directories or expand symbolic links, in contrast to the vnode architecture. The prefix mechanism completely replaces the UNIX mount mechanism, so servers are no more complex. The interface is optimized so that system calls usually require only a single server operation. The most important property is that the name space remains uniform across machines because it is the contents of the symbolic links at the mount points that defines how domains fit together, not a per-host or per-process configuration file. An final advantage with the prefix caching mechanism is that the root server is usually bypassed because clients quickly cache prefixes for the domains they use.

The primary limitation of the Sprite scheme is the use of broadcast to locate servers. This choice was made for simplicity, but it obviously limits the range of the name space. A general solution would be make an upcall in the case that the broadcast fails so that a user-level process can take arbitrary action to locate the server. For example, the Domain name server or some other name service could be used. This solution has the nice property that the kernel implements a lightweight mechanism (broadcast) that works for the common case, but can rely on the escape hatch to user-level in the hard case.

5 Maintaining State in a Distributed System

Another way to view the differences between the vnode and Sprite architectures is their support for maintaining distributed state. The Sprite architecture is inherently stateful, in contrast to the stateless file server model used in NFS, which was the main reason the vnode architecture was introduced. Sprite's stateful nature originally stemmed from its cache consistency mechanism that supports data caching on diskless workstations [Nelson88]. Delayed writes are used to optimize performance, but stateful servers keep track of how files are cached so they can guarantee UNIX file access semantics even when files are concurrently write shared. It turns out that server state is useful for a variety of other things.

Examples include the record of which host is executing a pseudo-device server, what files are open for execution so they cannot be overwritten, what devices are open in case they wish to enforce exclusive access, and file locking protocols.

The Sprite I/O interface has a number of entry points with no counterparts in the vnode interface, and these are almost entirely devoted to maintaining distributed state. The state does not have to be complex, it just has to be maintained carefully. It boils down to a per-object client list that records a few words of state about how that client is using the object. For most objects all that is required is the number open I/O streams to an object, expanded into counts for readers, writers, and executors. Shared and exclusive lock state is also kept in the client list, as well as type-specific state such as the current version number of files, or the ID of the host executing a pseudo-device server. For efficiency, a summary of all clients' state is kept in the server's object descriptor.

Maintaining state during normal operations is simple. State is initialized by the IoOpen and/or SrvOpen procedures. When handling RPC requests, the ClientVerify procedure is invoked to ensure that the server knows about the client. IoClose cleans up state about an I/O stream. This approach implies that the I/O server is contacted as a result of each open and close system call. However, I/O stream sharing that results from process creation does not require contact with the I/O server. This is an important optimization, and relatively easy to achieve by keeping a reference count on steam descriptors ("file descriptors" in UNIX terminology).

Life gets complicated by crash recovery and process migration [Douglis91]. A complete description of these mechanisms can be found in [Welch 90]. Only a few key points about these mechanisms will be made here.

Process migration results in open I/O streams that move around the network. This requires coordinated state updates on the original client, the new client, and the I/O server. This is complicated by multiple references to I/O streams that operate independently, yet share a common stream access position, or offset. When a process migrates, state about each of its I/O streams is packaged up and shipped to the process's new site as part of the process descriptor. However, no state changes are made when the process begins to leave. Instead, after the process is reincarnated on the new site a set of coordinated state changes occur. The new site notifies the I/O server for each stream, indicating that a stream reference has migrated to it. The I/O server makes a callback to the original client that retrieves the current stream offset and decrements the stream reference count there. The I/O server then updates its client list and replies to the new site. If there are no remaining stream references at the original site, then the new site can own the stream offset. Otherwise the server maintains the stream offset on behalf of both sites. This algorithm is somewhat delicate because there are often close operations that occur about the same time as migrations, so the locking order on data structures must be deadlock free.

Crash recovery is based on the following observations. First, the I/O servers keep their state organized on a per-client basis in order to support things like data cache consistency. This also makes it easy to clean up state about clients that crash. Second, it turns out to be straight-forward for clients to mirror the state that the server keeps about them. Recall that the state information is just counts of open I/O streams, plus type-specific state like file version numbers. The observation that clients can duplicate their server's state means that the system can recover from server crashes. The Reopen entry in the I/O interface is an idem-

potent operation that attempts to reconcile the client's state with the state that exists on the server. The success or failure of the Reopen call is dependent on the type of the underlying object and the actions of other clients. For example, a client can recover if it is writing to a file and has dirty blocks in its main memory cache, unless for some reason (i.e., a network partition) the server has allowed a different client to open the file and generate a conflicting version. Two features of this recovery scheme are worth repeating. First, the Reopen is idempotent so the client can invoke it whenever it thinks the server's state is out of date. Second, the server has the final word, and can always deny a reopen request.

6 Other Related Architectures

The Ultrix gnode interface [Rodriguez86] is quite close to the vnode interface. The rnode interface used in ATT Unix, however, shares some similarities with the Sprite architecture. Its architects classify it as a "remote system call" interface [Rifkin86]. Remote operations are trapped out at a relatively high level and forwarded to the remote node. For pathname operations in particular, this can result in fewer client-server interactions than a component-based interface. The implementation is a little gory, however. System calls are littered with checks against the remote case, and on the server side longjmp is used to warp execution back to the RPC stubs. This was done in order to avoid deeper structural changes required for a fully modular implementation. In contrast, the Sprite implementation is quite clean, which made it easy to add in new cases such as the notion of user-level servers.

The UIO interface of the V system is a clean design introduced to support Uniform I/O access in distributed systems[Cheriton87]. It is also coupled with a prefix table mechanism that is used to partition the name space among servers [Cheriton89]. However, the UIO interface also lumps naming and I/O operations together into one interface. The prefixes on pathnames are distinguished by a special character, ']', and so they just partition the name space among different classes of servers such as print, file, display, and tape servers. A global name service is used to map from prefixes to server multicast addresses, but each class of service still has to implement a name space for its objects. In contrast, the Sprite prefix mechanism transparently distributes a hierarchical name space among file servers, and the file system interfaces are designed to allow the file servers to function as more general name servers.

The Echo distributed file system developed at DEC SRC uses a more elaborate name service to transparently distributed a file hierarchy among servers. The system is complicated by support for replication, both at the name server level and at the file server level [Hisgen89]. In Echo, hosts still have local file systems, and the global file system name space is not used for remote device or remote service access.

The Plan-9 architecture [Pike 90], like Sprite, uses the file system name space to represent services, both kernel-resident and user-level. The details of the communication mechanisms are different, and the Sprite name space is fully shared via the prefix table mechanism, while the Plan-9 name space is per-process-group via the mount mechanism. Remote access is possible in Plan-9 by means of a server-server that can make connections to remote services. Sprite has additional mechanisms to support process migration and server crash recovery.

7 What I Would Do Differently

Perhaps the weakest point in the Sprite design is the handling of object attributes. Currently

this is distributed between the file server and the I/O server for the object. For files those servers are the same, but for devices and pseudo-devices they are different. Both the UNIX stat() and fstat() system calls result in RPCs to both the file server, to get most of the attributes, and the I/O server, to update attributes like the modify time. Also, the time stamps on the generic device files are not that well defined.

There are some more minor things that could be cleaned up. It turns out that the Release and Close operations are very similar, as are the IoOpen and MigEnd procedures. These operations maintain usage counts that reflect the number of read and write streams to an object. These four procedures could probably be replaced by a pair of slightly more general procedures, one to add a new stream, and one to remove state about a stream.

Finally, there are still some holes in the implementation that reflect lack of programming cycles. For example, while it is possible to migrate the client of a pseudo-device, migrating the server is a lot harder because the state of all the clients also needs to be updated. For some pseudo-device servers, like the X server, it doesn't really make sense to migrate the process to another host. With others, like the daemon that maintains state about idle hosts, it might be useful.

A more interesting missing feature is the ability of a pseudo-device or pseudo-file-system server to return a previously existing I/O stream in response to the IoOpen operation. Currently the IoOpen procedure only creates an Upcall channel to the pseudo-device server. However, the ability to pass back arbitrary streams would make it possible to implement archive file systems and version control file systems. The infrastructure that supports process migration makes it feasible to pass an I/O stream between any two processes, but integration with the pseudo-device IoOpen procedure just wasn't on any critical path so it didn't get fully implemented.

8 Conclusions

The main point that this paper makes is on the importance of cleanly separating the naming and I/O interfaces of the file system. This split is taken for granted in classical distributed systems literature that always includes a system wide name server [Wilkes80]. However, the distributed systems that evolved from UNIX implementations failed to incorporate this notion in the right way. SunOS, for example, uses a network database server to name hosts, password entries, and maps from file systems to file servers. However, once one enters the domain of a file system all things are tied to one host. Even distributed systems like V use a name service to partition servers at the top levels of the naming hierarchy. In Sprite the file servers generalize their directory structure mechanism to provide naming support for a variety of objects. The only things not named by the file system are hosts, which can be named via the Domain Name Service, users, and arbitrary processes (only pseudo-device servers have names in the file system name space).

9 References

Baker91. M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, "Measurements of a Distributed File System", *Proc. of the 13th Symp. on Operating System Prin., Operating Systems Review*, Oct. 1991, 198-212

Cheriton87 D. Cheriton, "UIO: A Uniform I/O System Interface for Distributed Systems", *ACM Transactions on Computer Systems* 5, 1 (Feb. 1987), 12-46.

Cheriton89 D. Cheriton and T. Mann, "Decentralizing a Global Naming Service for Improved

Performance and Fault Tolerance", *ACM Transactions on Computer Systems* 7, 2 (May 1989), 147-183.

Clark85. D. Clark, "The Structuring of Systems Using Upcalls", *Proc. of the 10th Symp. on Operating System Prin., Operating Systems Review* 19, 5 (Dec. 1985), 171-180.

Douglis90. F. Douglis, "Transparent Process Migration for Personal Workstations", PhD Thesis, Sep. 1990. University of California, Berkeley.

Hisgen89 A. Hisgen, A. Birrell, T. Mann, M. Schroeder and G. Swart, "Availability and Consistency Trade-offs in the Echo Distributed File System", *Proc. Second Workshop on Workstation Operating Systems*, Sep. 1989, 49-54

Nelson88. M. Nelson, B. Welch and J. Ousterhout, "Caching in the Sprite Network File System", *ACM Transactions Computer Systems* 6, 1 (Feb. 1988), 134-154.

Pike90 Rob Pike, Dave Presotto, Ken Thompson, and Howard Tricky, "Plan 9 from Bell Labs", Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, 1-9.

Rodriguez86 R. Rodriguez, M. Koehler, and R. Hyde, "The Generic File System", *USENIX Conference Proceedings*, June 1986, 260-269

Sandberg85 R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem", *USENIX Conference Proceedings*, June 1985, 119-130

Walker83 B. Walker, "The LOCUS Distributed Operating System", *Proc. of the 9th Symp. on Operating System Prin.*, Operating Systems Review 17, 5 (Nov.1983), 49-70

Welch86a. B. B. Welch, "The Sprite Remote Procedure Call System", Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

Welch86b. B. B. Welch and J. K. Ousterhout,"Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", *Proc. of the 6th ICDCS*, May 1986, 184-189.

Welch88. B. B. Welch and J. K. Ousterhout, "Pseudo-Devices: User-Level Extensions to the Sprite File System", *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.

Welch90. B. B. Welch, "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System", PhD Thesis, 1990. University of California, Berkeley.

Wilkes80 M. Wilkes, R. Needham, "The Cambridge Model Distributed System", Operating Systems Review 14, 1 (Jan. 1980).

An Object Oriented, File System Independent, Distributed File Server

Noemi Paciorek and Marc Teller[1]

Operating Systems Research Group
Center For High Performance Computing
Worcester Polytechnic Institute
293 Boston Post Road West
Marlborough, MA 01752

Abstract

Distributed file systems typically perform distribution in a file system dependent manner. The actual distribution mechanisms are usually embedded in file system dependent code. This paper presents a new distributed file server architecture, soon to be available in the OSF/1 AD system, that offers file system independent distribution mechanisms implemented in an object oriented manner. This server provides a single global name space with location transparency. As a benefit of the single name space support, the server preserves all of the UNIX semantics for accessing devices, both locally and remotely, unlike many other distributed file systems.

This server's architecture differs significantly from most distributed file servers, yet it offers a high degree of backwards compatibility, enabling file systems conforming to OSF/1 specifications to take advantage of the new distribution mechanisms with relative ease. This paper explores the server's architecture and contrasts it to other distributed file server architectures wherever possible.

1 Introduction

The CHPC distributed file server (DFS), a key component of the OSF/1 AD system, currently in Beta test, provides a new mechanism for file service distribution that is both object oriented and file system independent. Although the target system architecture for OSF/1 AD is a multicomputer, the implementation of the file service is architecture and network independent. In fact, the file service was initially developed on a cluster of workstations interconnected via Ethernet. The following paragraphs present both the motivations for and benefits resulting from the architecture and implementation of this file server. See [Levy90] for details on distributed file system concepts.

One of the more paramount considerations is the need to present a single file name space, with location transparency, in a distributed environment. In essence, the file name space looks exactly as if the user (or program) were viewing a non-distributed system. The need

1. The authors can be reached via InterNet at: noemi@chpc.org and mteller@chpc.org

for location transparency removes any requirement that the user have knowledge of the system configuration. In addition, multicomputer environments, in which many nodes cooperate to present a single system image, need to present a single name space.

In order to support file systems in existence today, and those that may be developed in the future, it was desirable to provide distribution mechanisms above the file system dependent code in the VFS layer. Adding a file system type to this distributed server does not require implementation of distribution mechanisms within the file system code itself; this is effectively inherited from the distributed VFS framework implemented above the file system. This also means that the distribution is compatible with the current OSF/1 paradigm and may easily be adapted to other VFS variants. Another benefit of preforming distribution above the file system dependent code is to relieve this code from the requirement of built-in knowledge of the communications mechanisms used to contact other servers. This built-in knowledge can make the file system code unduly dependent on the communications protocol.

Maintaining this file system independent approach allows significant code re-use from the OSF/1 VFS layer and the file system dependent code below it. This reduces the amount of risk since the OSF/1 file system code has been extensively tested. Maintenance of the server is equally simplified as most fixes and enhancements can be directly adopted into the distribution framework.

The next section details the evolution of the VFS paradigm and the features unique to the OSF/1 and CHPC versions. Subsequent sections present the file server architecture, management of file system independent objects, handling of mount points, and the servicing of file system requests. The paper concludes with a discussion of areas for further consideration and a summary.

2 VFS Paradigms

In many UNIX[1] compatible systems, the file system code is broken into two layers: the file system independent portion and the file system dependent code. Various models have been developed to provide generic file service, but the Virtual File System (VFS) paradigm has become the dominant model adopted by the major providers of UNIX-like operating systems (e.g. USL and OSF).

2.1 VFS Basics

The VFS layer manages file system independent attributes of files and file systems and calls into underlying file system code to perform the file system dependent operations. The VFS provides two generic abstractions: mounted file systems and files. Each mounted file system is described by a data structure (a *vfs* structure in Sun and AT&T kernels and a *mount* structure in BSD and OSF systems). Vnodes provide file system independent repre-

1. UNIX is a registered trademark of UNIX System Laboratories.

sentations of files; underlying file systems also maintain their own data structures to describe mounted file systems and file system dependent attributes of files.

The VFS layer interfaces with underlying file systems via VFS and *vnode* operations (VOPs). The VFS interface is used for operations on file systems (e.g. *mount*), while the vnode interface is used for operations on individual files (e.g. *open*). The vnode interface is used to bridge the generic and file system dependent portions of file naming and file access. In contrast, other systems such as Sprite [Welch90] and Amoeba [Tanenbaum89] separate naming and file access.

Throughout the years, several varieties of VFS architectures have developed, each with its own merits. The subsequent portions of this section describe the evolution of VFS paradigms, beginning with the SunOS model and concluding with the CHPC model.

2.2 SunOS VFS -- A Stateless Model

Sun developed the VFS paradigm to provide file service for both local and remote file systems [Kleinman86]. The introduction of the VFS layer necessitated the restructuring of system calls to operate on *vnodes* instead of *inodes* and to call into the underlying file system code via the vnode interface to perform file system dependent portions of operations. The UNIX file system (UFS) [McKusick84] providing local file service was then re-organized to conform to the new vnode and VFS interfaces and the network file system (NFS[1]) [Sandberg85] were introduced to provide remote file service. Thus, Sun provides two different file systems, one for local file service and another to manage remote files. The vnode and VFS operations for UFS are handled locally, while the NFS versions of these operations are forwarded to remote systems for service.

This VFS paradigm also provides per-file state on remote systems. Files remotely accessed via NFS are described by vnodes. In addition, NFS clients maintain *rnodes* to describe file system dependent attributes of remote files. Thus a file is locally described by a vnode and an inode and remotely via a vnode and an associated rnode.

Sun adopted a stateless paradigm for both its VFS and NFS implementations. In this model, the VFS layer maintains no state of its underlying file systems. Synchronization between I/O requests on files must be handled by the file system dependent code (e.g. via ILOCK for UFS systems). The VFS layer acquires no locks and makes no assumptions regarding the locking protocols, if any, implemented by the underlying file systems. This provides a very general interface and allows the file system dependent code to apply its knowledge of the file system in determining the locking structure that best suits it.

However, this does method does not guarantee the atomicity of all file system operations. Each stage of an operation is atomic, but subsequent stages of a multi-phase operation may not be. As an example, a file creation consists of two steps: pathname translation and file creation. It is possible for another process to create a file between the time the lookup

1. NFS is a trademark of Sun Microsystems, Inc.

is performed and the subsequent file creation stage. Therefore, to guarantee correctness, the directory to contain the file must be re-scanned prior to creating the new entry. In addition, this model requires that a multi-processor implementation re-scan hash chains to prevent duplicate insertions. Hence, the simplicity of the stateless model introduces the extra overhead of duplicated operations.

2.3 4.4BSD VFS -- A Stateful Model

The Computer Systems Research Group at UC Berkeley designed their own VFS paradigm to address the deficiencies of the stateless model [Karels86]. The Berkeley VFS provides a stateful model to guarantee atomicity of entire file system operations, including multi-stage ones. In this model, the VFS layer may instruct underlying file systems to acquire a mutual exclusion blocking lock (mutex) at any time via the VOP_LOCK vnode operation. Pathname translations use this facility to return locked vnodes to multi-stage system calls (e.g. *creat*). In this model, complex operations require locking both the relevant file and the directory it resides in. This lock is then held throughout the multi-phase system call and may be released by either the VFS layer or file system specific code after all the stages have completed. This mutex lock guarantees the atomicity of multi-stage operations and eliminates the duplicate work (e.g. directory re-scans) necessitated by the stateless paradigm.

However, the duplication is eliminated at the cost of serializing all directory operations. In this model, directories are always locked when they are scanned and remain locked whenever multi-stage operations are performed. This high degree of serialization is especially inappropriate for multi-processor (MP) systems. On an MP, it is even impossible to perform two read-only operations on a directory in parallel (e.g. performing **stat** on two files in the same directory). Also, some multi-stage operations may be very expensive in terms of time utilization and the directory remains locked during the entire multi-phase operation, even across blocking operations such as I/O. This may force read-only operations (e.g. **stat**) to block for the duration of entire multi-stage operations (e.g. rename). In addition, this model forces underlying file systems to adopt a particular locking protocol. Whenever the VFS layer calls into the underlying file system with a VOP_LOCK request, the file system code must acquire a mutex lock on its file system dependent data structure. This requirement is also inappropriate for multi-processor systems.

Thus, the stateful model is not well suited to multi-processor systems. However, it also provides a high degree of serialization for uni-processor systems because multi-stage operations do not release the mutex locks they hold upon blocking. Thus, read-only directory operations may be blocked by multi-phase operations that are also blocked. Hence, this paradigm eliminates duplicate operations at a cost that is extremely high for many architectures.

2.4 OSF/1 -- Stateless VFS Plus Timestamps

The OSF/1 VFS paradigm derives from the 4.4BSD model. However, the VFS layer in OSF/1 is stateless, while underlying file systems must either maintain minimal state or serialize all operations [LoVerso91]. The OSF VFS layer is fully parallelized, as well as its

UFS and NFS file systems. However, OSF/1 provides a model that is well suited to both uni-processor and multi-processor systems and to parallelized as well as un-parallelized file systems.

Unparallelized file systems are supported by binding them to a single processor, effectively serializing all their file system operations, while fully symmetric file systems may execute on any processor. Parallelized file systems use timestamps to eliminate virtually all re-scans inherent in the stateless model. Timestamps are monotonically increasing counters that track data structure modifications. They are incremented when a data structure is modified and examined during multiple stages of complex operations. If the timestamp changes between stages, the associated data structure has been modified and the earlier stage must be repeated.

Timestamps are used by the OSF/1 UFS file system to record modifications to directories. The example of creating a file illustrates their use. The pathname translation code in the VFS layer calls the UFS lookup operation which saves the timestamp of the directory where the file will be created. The subsequent phase of the creat operation allocates a new inode for the file, potentially blocking while reading it from disk. In the final phase of the operation, a directory entry is created for the new file. At this point, the UFS layer compares the current parent directory's timestamp with the value saved during the pathname translation. If they differ, the directory has been modified since the lookup and, therefore, must be re-scanned to ensure that the file has not been created. If the timestamps are identical, the directory entry can simply be created because the directory has not been modified since the pathname translation on the file. (See [LoVerso91] for further details).

Thus, the OSF/1 model exhibits the benefits of both the stateless and stateful models. Its stateless VFS paradigm is also more general than the stateful model. The generic nature of this VFS paradigm has been proven by its support of several file systems, including AFS[1] [Spector89] and Episode [Chutani92]. In addition, the OSF/1 VFS also provides the benefit of fully symmetric operation and support for both parallelized as well as unparallelized file systems.

2.5 CHPC VFS -- Distributed and Stateless

The three VFS paradigms described thus far support both local and remote file systems, but do not easily lend themselves to supporting file systems distributed across multiple computers. In addition, all support for remote file systems lies within the file system dependent code. Thus, a vnode interface call (VOP) may cause an operation to be performed locally or may issue a remote procedure call or send a message to another system requesting it to perform the function.

The CHPC VFS paradigm was developed to provide support for a location transparent single name space, both in a multi-computer, as well as in more traditional architectures. This VFS layer provides distributed name service coupled with direct access to remote file

1. AFS is a trademark of Transarc, Inc.

servers for other vnode operations. The distributed aspects of the name space management are transparently handled by the VFS layer instead of by the file system dependent code, providing a file system independent distribution mechanism. This allows a single file system type to service both local and remote files and also allows previously undistributed file systems, such as UFS, to take advantage of the distribution mechanisms in the VFS layer. Thus, the single name space may be supported by a single type of file system distributed across many nodes.

The CHPC VFS layer also provides direct remote access to files. In the VFS models presented above, access to remote files is supported via the vnode interface. The various VOP calls of the previous models result in remote procedure calls when operating on remote files. In the CHPC VFS model, VOP calls are always serviced locally because operations on files are always directed at the file server managing the file. (Later sections of this paper describe this mechanism in great detail.)

Further, the CHPC VFS paradigm does not maintain per-file state on remote nodes. There is no need for the equivalent of an rnode in this model because all access to remote files is performed directly, by sending messages to the appropriate file server, instead of indirectly via VOP calls. This allows a local file system, such as UFS, to be distributed without maintaining additional state. Again, subsequent sections of this paper describe this scheme further.

Hence, the CHPC VFS model provides all the functionality of the OSF/1 paradigm with additional features to support file system independent distribution of the name service as well as direct access to remote files. This has been achieved without modification to the mechanisms that support local, non-distributed file systems.

2.6 Comparison With Other Models

The VFS model for distribution has evolved significantly over the years. The CHPC paradigm differs in many ways from the original model created by Sun. The rest of this section briefly examines the differences between the CHPC distributed file server and other VFS based file servers and also provides a general comparison between the VFS model and schemes developed by other distributed file systems.

2.6.1 CHPC VFS Versus Other VFS Based File Systems

The CHPC VFS paradigm transparently supports both local and remote file systems. The VFS layer performs distributed name service for file systems that do not provide their own distribution mechanisms. Thus, a single name space is created via VFS based distribution of a local file system such as UFS. However, the CHPC DFS also supports file systems that provide their own distribution protocols. The first release contains support for NFS and AFS support is viewed as desirable. However, these file systems do not enforce a single name space and are supported solely for compatibility with other systems.

In particular, the NFS protocol allows servers to export selected directories to their clients. These clients, however, are free to mount the exported directories as they choose, poten-

tially creating disparate views of the shared portions of the name space. This model does not lend itself to support for a single name space. In addition, NFS does not provide remote device access. The CHPC DFS, on the other hand, allows special files to represent remote devices and provides full UNIX semantics for access to these devices via their remote special files.

RFS provides access to remote devices co-located with their file servers [Rifkin86]. However, the CHPC DFS provides transparent access to remote devices on any node, even those that do not provide file service. RFS servers advertise selected directories to their clients, as do NFS servers, thus allowing clients to mount the directories in a random fashion. In addition, RFS allows servers to subdivide the name space by providing a domain based naming scheme. Thus, RFS is extremely ill-suited for single name space support. Further, RFS, like NFS, does not offer direct access to remote files; they are accessed indirectly via state maintained by the client.

The AFS distributed model has also gained popularity in recent years as a wide area file system. The AFS name space consists of a potentially large shared name space managed by servers in addition to private name spaces resident on client systems. Thus, AFS provides the ability to share part of the name space while also maintaining local file systems on separate disks. This file system works well in networks of distributed workstations with disks but does not easily lend itself to other architectures. AFS, like NFS and RFS, was also not designed to support a single namespace.

The CHPC model, however, does provide a single name space. Yet it is also capable of supporting file systems which perform their own distribution and do not offer single name space support. In addition, the CHPC DFS also provides distributed device support.

2.6.2 Comparison With Non-VFS Architectures

Several recent distributed systems are not based on the VFS paradigm. In particular, Sprite and Amoeba have adopted models very different than those provided by the VFS. Both of this systems provide separate mechanisms to offer name service and file access while the VFS paradigm provides both.

The Sprite distributed system offers a distinct separation between naming and I/O services. Opening a file in Sprite consists of pathname translation followed by opening an I/O stream. This separation also provides the means for Sprite to support remote devices [Douglis91] and has allowed Sprite to optimize the common case of regular file opens. (See [Welch91] for further details).

The Amoeba file system consists of three servers: the directory server provides naming and directory management; the bullet server provides file storage; and the replication server manages file replication. This file system also differs from others by only providing support for immutable files. This simplifies the file and replication servers immensely. (See [Tanenbaum92] for further details).

Although the VFS paradigm is the most widely spread model for providing distributed naming and file access in UNIX compatible systems, some recent distributed systems have

created efficient distribution models by other means. However, the CHPC DFS has maintained a great degree of compatibility with OSF/1 file systems by evolving the VFS layer to provide support for new features including a file system independent distribution mechanism, a single global name space, and remote device access. Subsequent sections of this paper provide much greater detail on these new features.

3 Distributed File Server Architecture

The architecture for the distributed file server builds upon the UNIX mount point model, using mount operations on both local and remote file systems to assemble a single system-wide name space. Each file system is managed by a unique file server which, in turn, is likely to manage more than one file system. In the case of remotely mounted file systems, the mount point crosses a node boundary from one file server to another. To prevent this from adversely impacting performance, the mount list is distributed with each file server maintaining mount structures for the file systems it manages.

This distribution extends to file system independent objects (e.g. files, directories, etc.) managed by the server. These objects and their associated state are spread across the name space. Whenever possible, the CHPC DFS distributes state that has typically been centrally maintained to avoid bottlenecks. The file server distributes the state associated with open files and mounted file systems to the servers that manage the objects.

In particular, each open file is described by an entry in an open file table. This table is distributed so that each server only contains entries for the files it services. Therefore, the seek pointer (for example) is maintained by the server and not by the client as in traditional distributed file systems such as NFS.

Since the state is maintained by the server managing an object, no per-object state must be maintained on remote nodes and the file service provides direct access to remote objects instead of the traditional indirect access via the vnode interface. Refer to Section 6.2 for more information on direct file access.

The path from client to server may involve any number of other file servers as the pathname is traversed. The forwarding from one server to the next is transparent and occurs when a remote mount point is encountered. Once the file has been opened, all further access follows a direct route, as mentioned above.

Remote devices may be accessed without regard to their location in the system. The server transparently maps special files to their remote devices without creating new file types in the name space. Full UNIX semantics for device access are preserved and clients need not have any knowledge of the true location of the devices in the system. Unlike regular files, however, special files are handled in a unique manner. An *open* operation is handled by two file servers: the server managing the special file (usually located in /dev), and the server managing the device referred to by the special file. In a similar manner, the last *close* operation on the device is handled by the same two servers.

The architecture for the server is based upon a message passing environment, Mach IPC, which provides media and protocol independence [Draves90]. As a result, file systems and the VFS layer do not require any knowledge of the protocol used to communicate with clients and other servers. This also allows file systems conformant to the VFS paradigm to be distributed without providing their own distribution mechanism.

The server communicates with its clients via Mach ports, which serve as the end-points for communications channels. Another feature of Mach IPC the server uses is the *no-more-senders* notification, which automatically informs the server when no more tasks are actively using a port. This mechanism provides the means for port garbage collection and relieves the server of the burden of reference counting senders on its ports. See [Draves90] for further detail on *no-more-senders* and ports in general. Although this server is built upon Mach IPC and utilizes the features provided, it could be ported to another message passing paradigm. Thus, the subsequent sections of this paper focus on the distribution aspects of the server and not on its communications model.

4 File System Object Management

Each file system independent object managed by the server is represented by a Mach port which hides the locality of the object and provides a level of security via the capabilities associated with the port [Draves90]. Table 1, below, presents these representations. As shown, the file server may return a send right to a port representing a file system independent object.

TABLE 1. File System Object Representation

File System Object	Representation
File	Vnode Port
Open File	File Structure Port
Directory	Root or CWD Port
Mount Point	Mount Structure Port
Special File	Vnode Port

File Objects

Files are represented by a vnode port which is associated with the file system independent representation for a file (i.e. the vnode).

Open File Objects

When a file is opened, its representation changes from that of a vnode port to a file structure port, which corresponds to the data structure containing the current state of activity on the open file (i.e. the file structure).

Directory Objects

Directories are viewed as a special instatiation of a file, so they are also represented by vnode ports. These ports, however, have a special meaning consistent with the UNIX semantics associated with the root directory (top of the file system name space) and the

current working directory (default directory for a process' file activity). Therefore, the ports have unique names: root port and cwd port.

Mount Point Objects
Mount points are represented by a port associated with the data structure containing information used to manage the file system name space assembly and disassembly. More on the mount point paradigm can be found in Section 5.1.

Special File Objects
Special files referring to devices are represented by vnode ports. The CHPC distributed file server handles special files representing remote devices and uses vnode ports to provide access to both remote special files and the devices they refer to. See [Paciorek92a] for details on this implementation, as the discussion is beyond the scope of this paper.

5 Mounting File Systems

As mentioned earlier, the CHPC distributed file server maintains a global name space via a mount point model. This section details how the mount point model is used to transparently join file systems together to create the single name space.

5.1 CHPC DFS Mount Paradigm

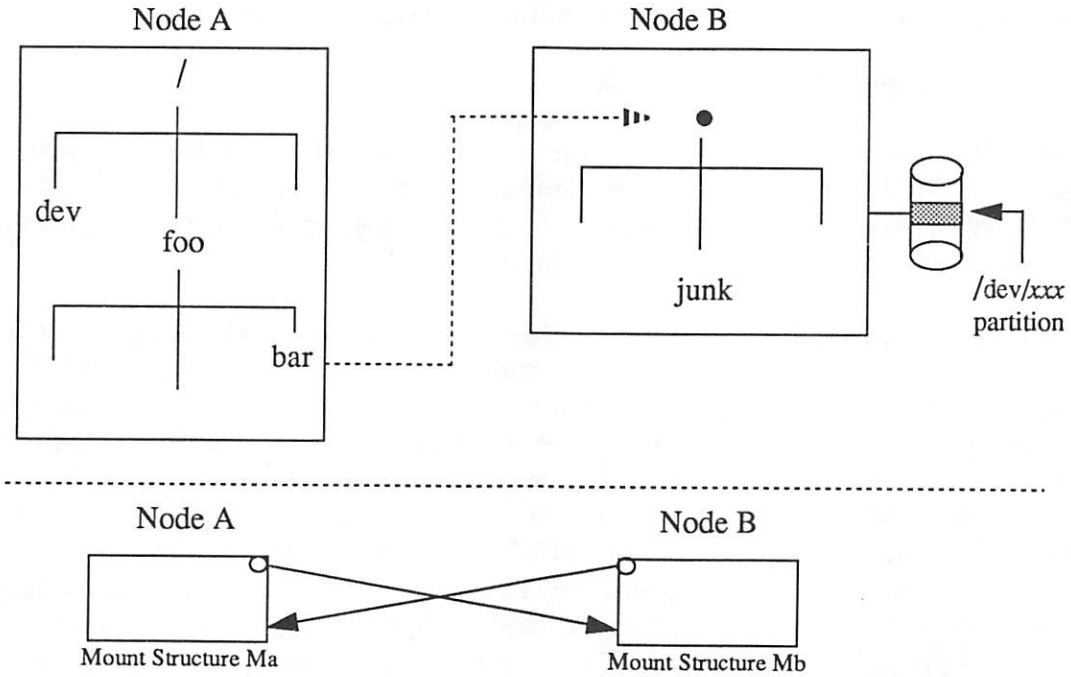
The OSF/1 mount point model was easily extended to describe both locally and remotely mounted file systems because both types of mount points are indistinguishable from each other in the name space. Both are simply represented by directories. The file server also uses the same data structures to describe both types of mounted file systems.

In the OSF/1 VFS layer, mounted file systems are represented by mount structures and their files are represented by vnodes. The lookup code translates components of pathnames to vnodes. It also uses mount structures to locate the root vnode for a file system when traversing a mount point from above and the vnode for the parent directory of the mount point when translating ".." at a mount point. In the CHPC VFS layer, each mount point representing a remotely mounted file system is described by two mount structures: one on the node managing the mounted-on directory and one on the node servicing the file system. Each of the mount structures contains a send right to the mount structure port representing its remote mount structure counterpart. The server forwards file system operations to the appropriate mount structure port when crossing remote mount points. The server's use of ports, including mount structure ports, is discussed in Section 4. Also, see [Paciorek92b] for a more detailed description of the server's management of ports.

An example best illustrates the use of mount structures and ports. Figure 1 depicts a remote file system on Node B described by a special file, "/dev/xxx" on Node A. This file system is mounted onto the "/foo/bar" directory on node A. Thus, the mount point "/foo/bar" describes a remote file system and the special file "/dev/xxx" describes a remote device. The mount point is represented locally by mount structure Ma on node A

and via mount structure Mb on node B; it is also accessible remotely via mount structure ports on both nodes.

FIGURE 1. Remotely Mounted File Systems



The pathname translation of "/foo/bar/junk" crosses a remote mount point. When namei arrives at the "bar" directory on node A, it detects that "bar" is a mount point representing a remote file system and extracts the port for mount structure Mb from mount structure Ma. The file server later forwards the system call message to port Mb and the pathname translation continues on node B starting with the root of the file system. This is described in great detail in Section 6. The translation of "../.." at "/foo/bar/junk" is handled similarly. When namei attempts to translate ".." at the "/foo/bar" directory, the lookup must cross from node B into node A. Namei acquires the port for mount structure Ma from mount structure Mb and forwards the system call to port Ma. The server managing node A, services the system call and continues the pathname translation.

Thus, the CHPC model extends the mechanisms OSF/1 uses to describe mounted file systems in a logical and consistent fashion. It does, however, have one drawback: most file systems conforming to the OSF/1 VFS paradigm require changes to the file system dependent mount code to take advantage of the distribution mechanisms in the VFS layer. But this is the only portion of the file system dependent code that must be modified to take advantage of the VFS based distribution scheme; no new vnode or VFS operations must be written. The type of changes required to the mount code and the difficulty of the implementation vary with the type of file system and will likely be lower in future releases of the CHPC DFS. Modifications are required because the file system dependent portion of

the mount system call is very difficult to generalize. However, all the changes to the unmount code are localized to the VFS layer. Although the CHPC mount paradigm requires some support in the file system dependent code, the amount of code required is relatively small and much less work than writing new vnode and VFS operations to provide distributed support for file systems as other VFS models mandate.

5.2 Comparison with Other Models

The CHPC DFS transparently handles remotely mounted file systems in a single name space without creating new file types. However, both the Sprite [Welch91] and Chorus [Armand89] distributed systems have implemented a single global name space by creating new file types to describe remote mount points.

The Chorus distributed system creates a global name space by representing remote mount points in the name space by symbolic port names. A symbolic port name is a new file type that associates a file name to a unique Chorus port identifier. The port represented by one of these new symbolic links designates the file manager that services the remote file system. The pathname translation code has been extended to interpret symbolic port names and forward the pathname translation to the server it represents. Thus, the Chorus approach requires a new file type and changes to the file system code to manage it. Also note that although Chorus uses ports, they represent servers and not objects managed by servers. Thus, the Chorus mount paradigm differs from the CHPC one in two major ways: it requires the creation and support of a new file type and it uses ports in a less object oriented fashion than the CHPC DFS.

Sprite's support for a single global name space is similar to the Chorus model. A special type of symbolic link represents a mount point in the name space. This symbolic link simply contains the absolute pathname of the mount point. Sprite uses a broadcast to locate the actual server for the file system and then accesses the server via a kernel-to-kernel remote procedure call protocol. The remote server then continues the pathname translation. The Sprite mount paradigm also differs from the CHPC mount model in many ways. The most basic difference is that Sprite uses a RPC protocol instead of message passing and, locating remote servers is more difficult than in the CHPC DFS or Chorus system. Sprite's use of broadcasts to locate remote file servers also creates more overhead than representing remote file servers and their objects by ports.

The CHPC mount paradigm is implemented quite differently than those of Sprite and Chorus, yet it offers a very consistent and logical extension of the local mount model without introducing new objects into the name space. It also allows file systems to take advantage of the file system independent distribution mechanisms without significant code modifications.

The next section describes how the server handles file system requests directed at both local and remote objects, including its use of mount structure ports when transparently forwarding system call requests.

6 Servicing File System Requests

The CHPC distributed file server employs the same model as the OSF/1 file system for managing local requests. However, it also transparently forwards system call requests when operating on remote objects. This section elaborates on the server's forwarding scheme, but first it presents a brief description of each of the file service layers and its function.

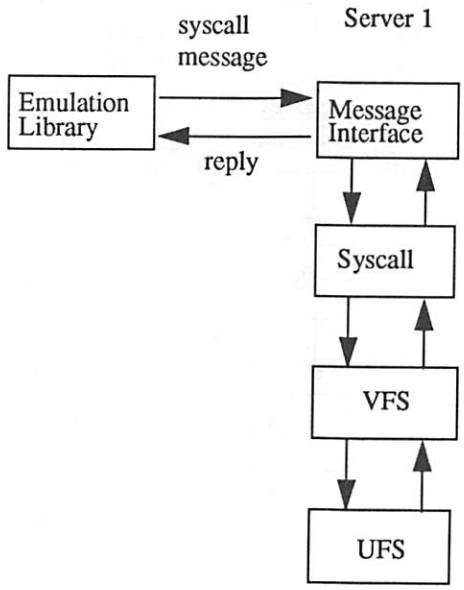
6.1 File Service Layers

The CHPC distributed file server utilizes the same layering scheme as the OSF/1 file system, with the addition of a new Message Interface (MI) layer that receives messages from clients and sends back replies. Figure 2 depicts the layers. Upon receipt of a message, a thread in the MI layer performs the following:

- Registers itself as servicing the request.
- Initializes its credentials before providing service on behalf of its client.
- Translates any ports in the message to the data structures that represent the associated objects locally.
- Performs any additional data structure initialization required by the message. (e.g. requests sent to vnode ports must set up additional state required by the pathname translation code).
- Invokes the appropriate system call.

While servicing the request, the syscall layer may invoke the pathname translation code in the VFS layer to lookup files and the VFS layer, in turn, calls into the underlying file system to perform file system dependent aspects of the translation. Eventually the system call completes and the MI layer returns the appropriate reply to the client.

FIGURE 2. Local File Service Operations



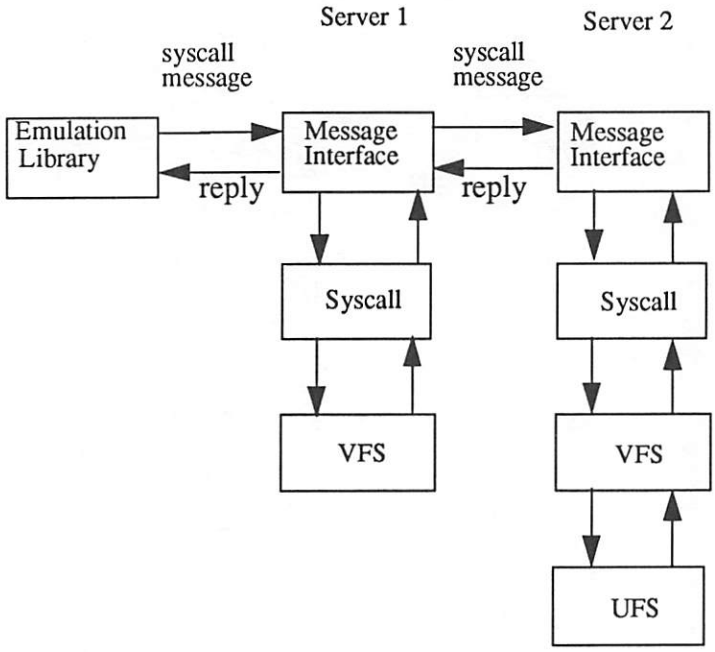
6.2 Operation Forwarding and Direct Remote Access

The CHPC distributed file server handles all file system requests locally until it must forward the operation to a remote server. Two events cause the server to forward system call operations: pathname translation crossing a remote mount point and opening a special file that represents a remote device. This section presents details on the process of forwarding requests when crossing remote mount points. Opens of special files representing remote devices are handled analogously.

The OSF/1 namei function detects when a pathname translation crosses a remote mount point. The CHPC version of the function also distinguishes remote mount points from local ones by examining the flags field in the mount structure representing the mount point. When it encounters a remote mount point, namei obtains the remote mount structure port from the local mount structure and returns an error. Thus, the underlying file system code is not invoked when pathname translation arrives at a remote mount point, as depicted in Figure 3. After receiving the error returned from namei, the invoking system call also propagates the error back to the MI layer, which then forwards the original system call request to the remote mount structure port that namei saved. Before sending the message, however, the MI layer replaces the original pathname with the untranslated remainder of the pathname.

A file server thread on the remote node receives the message and invokes the system call after completing any required preparation. The system call then invokes namei to continue the lookup of the remainder of the pathname. Once again, namei may detect a remote mount point and the procedure described above is performed until the translation resolves locally on some node, as shown in Figure 3.

FIGURE 3. Remote File Service Operations



Once the file has been located, the local file server thread continues to perform the rest of the system call, including any vnode operations (e.g. VOP_OPEN). Thus, there is no need to send messages to remote servers to perform vnode operations because the file is serviced directly by the file server managing the file system it resides in. Once the system call completes, the server replies to the original client.

The example of an open system call described below illustrates the server's handling of file system requests and operation forwarding. The steps involved in opening a file include the following:

- A client sends an open message to its root or current working directory vnode port.
- A thread in the file server's MI layer receives the message and performs any required preparation before invoking the open system call.
- The open code calls namei to perform pathname translation.
- The translation may cross mount points. If namei detects a mount point represents a remote file system, it returns an error back to the open code.
- The open system call code propagates the error back to the MI layer, which in turn forwards the open system call, including the remainder of the pathname to be translated, to the mount structure port on the remote node.
- A file server thread on the remote node receives the message and invokes the open system call after completing any necessary preparation.
- The open code calls namei to complete the rest of the pathname translation. Again more remote mount points may be crossed, causing the forwarding process to be repeated.
- Eventually a file server thread resolves the pathname translation locally and completes the rest of the open system call.
- The resolving thread allocates a file structure and an associated port to represent the open file, returning the port to the original client.
- The client then accesses the file directly via the file structure port and directs any requests (e.g. read) on the open file to that port.

Thus, the CHPC model provides direct access to remote files once the distributed naming services in the VFS and file system dependent code have located the file. Hence, remote file servers do not maintain state (e.g. vnodes) for remote files because the files are always accessed directly.

7 Areas For Further Consideration

The first phase of the CHPC distributed file server has been completed and the server is fully functional. The next stage, to begin shortly, will address issues of performance, scalability, and resource utilization.

Several obvious performance improvements will be made, including client [Julin91] and server side [Welch86] prefix tables, client caching of data, and server port caching. Further, to increase scalability, the server must provide replication in a file system independent manner. In addition, the server must support very large file sizes and distributed file

striping [Cabrera91] for super-computer applications. File system independent support for an extremely fast I/O path via new vnode operations is currently being designed.

Another area to address involves optimizing the utilization of some resources to eliminate the potential for starvation. Also, the special files code, based on the OSF/1 model, should be revisited and simplified.

The file server currently supports full UNIX semantics, unlike other distributed file systems such as NFS and AFS. However, many improvements will be made in future releases to provide increased performance and scalability.

8 Summary

The CHPC distributed file server offers a novel approach to file system distribution by providing an architecture independent and file system independent distribution mechanism. It offers a global name space and provides direct access to remote file system independent objects within the confines of the VFS paradigm. The architecture also provides all the UNIX semantics for remote device access via the same VFS mechanisms. The extensions to the OSF/1 VFS model are logical and consistent, yet allow easy incorporation of file system dependent code.

This distributed file server is a further step in the evolution of VFS based distributed file systems. It provides features not readily available in other distributed file systems built around the VFS model. However, the server needs further development to meet the increasing demands for performance and scalability.

9 References

[Armand89] F. Armand, M. Gien, F. Herrmann, and M. Rozier, "Revolution 89 or 'Distributing UNIX Brings It Back to its Original Virtues'", *Proceedings of the Usenix Workshop on Distributed and Multiprocessor Systems*, October 1989, pp. 153-174.

[Cabrera91] L. Cabrera and D. Long, "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates", *Computing Systems*, Vol. 4, No. 4, Fall 1991, pp. 405-436.

[Chutani92] S. Chutani, O.T. Anderson, M.L. Kazar, B.W. Leverett, W.A. Mason, and R.N. Sidebotham, "The Episode File System", *Proceedings of the Winter 1992 Usenix Conference*, January 1992, pp. 43-60.

[Douglis91] F. Douglis, J. K. Ousterhout, M. F. Kaashoek, and A. S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite", *Computing Systems*, Vol. 4, No. 4, Fall 1991, pp. 353-384.

[Draves90] R. Draves, "A Revised IPC Interface", *Proceedings of the Usenix Mach Workshop*, October 1990, pp. 101-122.

[Julin91] D. P. Julin, J. J. Chew, J. M. Stevenson, P. Guedes, P. Neves, and P. Roy, "Generalized Emulation Services for Mach 3.0 Overview, Experiences and Current Status", *Proceedings of the Usenix Mach Symposium*, November 1991, pp. 13-26.

[Karels86] M. J. Karels and M. K. McKusick, "Toward a Compatible Filesystem Interface", *Proceedings of the Autumn 1986 EUUG Conference*, September 1986.

[Kleinman86] S. R. Kleinman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of the Summer 1986 Usenix Conference*, June 1986, pp. 238-247.

[Levy90] E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples", *ACM Computing Surveys*, Vol. 22, No. 4, December 1990, pp.321-374.

[LoVerso91] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg, "The OSF/1 UNIX Filesystem (UFS)", *Proceedings of the Winter 1991 Usenix Conference*, January 1991, pp. 207-218.

[McKusick84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[Paciorek92a] N. Paciorek and M. Teller, "Design Specification For An OSF/1 Based Distributed File Server On Mach 3.0", Technical Report TR92-001R, Center For High Performance Computing, January 1992.

[Paciorek92b] N. Paciorek and M. Teller, "A File System Independent Distributed File Service for Mach 3.0", Technical Report TR92-002, Center For High Performance Computing, March 1992.

[Rifkin86] A. P. Rikfin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, "RFS Architectural Overview", *Proceedings of the Summer 1986 Usenix Conference*, June 1986, pp. 248-259.

[Sandberg85] R. Sandberg, D. Goldber, S. Kleinman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem", *Proceedings of the Summer 1985 Usenix Conference*, June 1985, pp. 119-130

[Spector89] A. Z. Spector and M. L. Kazar, "Uniting File Systems", *Unix Review*, March 1989.

[Tanenbaum89] A. S. Tanenbaum, R. van Renesse, H. van Staveren, and G. J. Sharp, "Experiences with the Amoeba Distributed Operating System", Technical Report IR-194, Department of Mathematics and Computer Science, Vrije Universiteit, July 1989.

[Tanenbaum92] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.

[Welch86] B. B. Welch and J. K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", *Proceedings of the 6th ICDCS*, May 186, pp. 184-189.

[Welch90] B. B. Welch, "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System, PhD Thesis, 1990, University of California, Berkeley.

[Welch91] B. Welch, "The File System Belongs in the Kernel', *Proceedings of the Usenix Mach Symposium*, November 1991, pp. 233-249.

DataMesh research project, phase 1

John Wilkes
Hewlett-Packard Laboratories
wilkes@hpl.hp.com

Work performed jointly with Chia Chao, Robert English, David Jacobson, Sai-Lai Lo, Chris Ruemmler, Bart Sears, Alex Stepanov and Rebecca Wright

1 Introduction

This position paper describes work taking place in the DataMesh research project [Wilkes89, Wilkes91] on *concurrent file systems*. A concurrent file system exploits parallelism in its construction, while presenting an external image that is compatible with existing client:server interface definitions. The work is described in a number of stages:

- the DataMesh hardware architecture is introduced;
- Jungle, the overall software architecture framework, is described;
- some existing results are described to demonstrate progress so far.

The DataMesh project is the overall umbrella activity for the research we are carrying out in the area of concurrent file systems. We believe our work is of considerable interest to future file system designers, as well as people interested in understanding and modifying existing file system designs.

The first phase of the DataMesh project is providing block-level services to its clients: the interface is in the form of read/write operations on sets of fixed-size data blocks. (You might like to think of this as the regular SCSI disk command set.) Phases 2 and 3 will provide interfaces at the file and record level respectively. Our design center is for a single DataMesh to contain perhaps 100–200 nodes.

We acquired our first DataMesh hardware preprototype in the summer of 1991 (we plan to replace it with something more closely resembling the scale and performance of our 1995 system over the next year and a half); current work emphasizes the development and evaluation of our software ideas using it. The work described here is in the development stages and doubtless subject to modification as we go along.

2 The DataMesh hardware architecture

Our goal is to develop a system storage architecture to provide high performance, high availability, scalability (in both size and component type), and standards-based connections to the open systems environment. We believe that these requirements in turn suggest particular solutions:

- *high performance*: the use of parallelism and the close coupling of processor power with storage elements;
- *high availability*: no single points of failure (i.e., there must be built-in redundancy), coupled with fault tolerant software;
- *scalability and long life*: a modular architecture, to allow a DataMesh server to expand and adapt to changing requirements over time, with smooth incremental growth;
- *open systems interconnection*: the ability for a DataMesh server to attach to the outside world through several hardware and software interconnect standards.

Our chosen hardware solution is an array of nodes of various types:

- *port nodes* provide connectivity to the outside world through an I/O interface like SCSI or a LAN like FDDI;
- *disk nodes* for storage;
- *RAM nodes* (volatile or non-volatile) for caching, read-ahead, and write-behind;

– and *tertiary storage nodes* (e.g., an optical jukebox, R–DAT tape, or robot tape library).

The nodes are linked by a fast, reliable, small-area network, and programmed so that they all cooperate to appear as a single storage server to its clients.

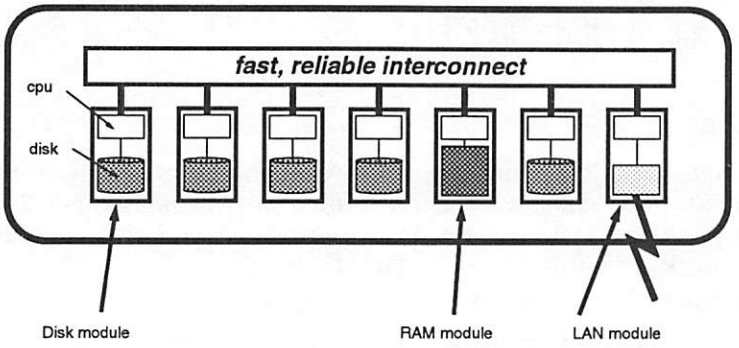


Figure 1. *System hardware model.*

By about 1995, we believe that current trends in silicon process and magnetic recording technologies will mean that each (low cost) disk node will comprise a dedicated 20 MIPS single-chip processor with about 16 MB RAM—as the disk controller—and a 2GB 3.5″ disk drive mechanism. The inter-node interconnect will be capable of 10μs round-trip latency and 10MB/s bandwidth per node, both scalable up to several hundred nodes.This interconnect will be highly fault-tolerant by virtue of a great deal of built-in redundancy. And the resulting system will cost almost the same as a simple array of regular disks with similar capacity. Existence proofs for all these statements (except the cost!) can be found in research prototypes or commercial products today.

3 The Jungle software architecture

> **Jungle**, *n.* (Area of) land overgrown with underwood or tangled vegetation, especially in tropics; scene of ruthless struggle for survival; wild tangled mass. [from Hindustani *jangal* from Sanskrit *jangala* desert, forest]
>
> — *The Concise Oxford Dictionary of Current English, Oxford University Press, 6th Edition, 1976*

The overall DataMesh software architecture is called Jungle. It is designed to exploit the replicated, distributed hardware, the low latency interconnect, and the dedicated processing power in a DataMesh server. As a result, much of the Jungle architecture is concerned with things like maintaining data coherency for correctness, replication and redundancy for fault tolerance, and local caching for fast access.

Since Jungle was intended as a framework into which a number of specific algorithms and policies can be fitted, the novel ideas it contains are more to do with managing collaborating components than with particular policies. Here are a few of the things that we consider important about it:

- Encapsulation of policy decisions in *managers*. Several different policies can co-exist, with each manager specialized to serve one set of needs optimally.

  The policies will optimize for different use patterns or storage needs. For example: multiple file access methods can present the same external interface, but one will be optimized for groups of small files, another for very large ones that are always accessed sequentially. Similarly, extensible byte vectors can be implemented in many different ways: e.g., fixed-size page allocation, or a buddy-system-based extent map.
- Jungle will provide system-wide management of physical resources such as RAM caches and storage devices.

For example, cache RAM in any of the Jungle nodes is treated as part of a global pool to be allocated to claimants, whose needs are assessed (and responded to) on a system-wide basis.

- Jungle supports descriptions of performance and availability properties of storage devices and the abstractions developed on top of them, such as files. With this we are able to provide mechanisms to match user-specified needs to available storage resources (a description of our approach may be found in [Wilkes91a]). Sample applications include selective high-availability and multi-media performance guarantees.
- Upper layers will be able to extend the functionality of lower ones by downloading interpreted scripts that express the upper-layer policies, but execute in an environment that is tightly bound to the data—thereby avoiding unnecessary and costly protection and machine boundary crossings.
- Jungle provides explicit handles (called *operation groups*) on the sequencing and completion properties of related sets of function calls.

3.1 Overview of the Jungle structure

The underlying model of Jungle is of a low-level, smart *chunk store* that holds raw bags of bytes, on top of which there is a layer of *chunk vector managers* that provide an abstraction composed of sequences of chunks. Finally, there is a layer of *thing managers* that provide application- and system-specific interfaces to the stored data.

Jungle software runs on both DataMesh server and client nodes. Figure 2 shows the layer structure of the major Jungle components, without making visible the hardware boundaries. In practice, the "upper" levels will reside on the workstations, the "lower" ones in the DataMesh server itself. The objects manipulated inside Jungle are:

- *Devices*: suppliers of *slots* where data can be stored. All the slots in a single device have the same set of attributes (performance, availability, cost, etc.). A single logical device may be constructed from one or more physical storage elements (e.g., a disk array); can potentially perform replication; and can choose to hide multiple layers in the storage hierarchy (e.g., an optical jukebox front-ended by one or more disk drives).
- *Chunks*: individual pieces of storage to be stored in device slots. (Chunks are our abstractions of the "blocks" commonly found in file systems.) A chunk has a fixed size over its lifetime, determined largely by what is optimal to store on the physical medium with which it is associated; different chunks may have different sizes.
- *Chunk Vectors*: sequences of contiguously-addressed chunks. Every chunk belongs to a single chunk vector. Chunk vectors can be extended only by whole chunks: they are like a very low level UNIX file abstraction.[1] Chunk vectors may provide or be associated with performance

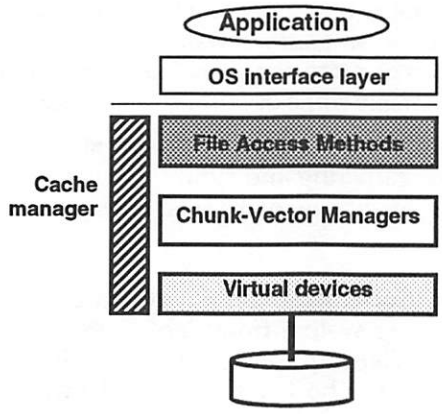


Figure 2. DataMesh *software architecture—simplified logical layering*

and availability properties, such as "always accessed in sequence"; "minimum aggregate bandwidth of 5MB/s"; "maximum 30 second downtime".

- *Upper Level Jungle Things (ULJT)*: application-visible objects with byte- or record-level interfaces. Access to these functions are provided by *Jungle-Thing Access Managers (JTAMs)*—also called "(file) access methods". Examples of ULTJs are UNIX files, databases, key-sequenced files, and persistent objects managed by a language runtime system.

Several implementations of each component can be active and available concurrently. For example, there will be chunk vector managers optimized for different performance properties: UNIX-file JTAMs optimized for different file sizes will coexist—one for many small files per chunk vector; one for single file per chunk vector (rather like existing UNIX file systems); and one for extremely large files that need to span multiple chunk vectors.

In addition, the Jungle architecture provides support for the following secondary objects:

- *Caches*: RAM memories that can be used to store chunks in low-latency memory. Cache managers respond to requests for physical memory from other managers, trying to satisfy those that are most "meritorious". (We plan to explore different algorithms for expressing and calculating such merit information.)
- *Operation groups*: sets of invocations of Jungle functions that should be considered together in some fashion for performance or availability reasons. For example: "do these in this order", "all these should be atomic"; "sequencing within this group is unimportant"; "*this* group should all happen before any of *that* group".
- *Scripts:* small interpreted programs that the upper levels of Jungle use to augment the functionality of lower layers. Once a script has been downloaded, it acts as a subroutine that can be called. Scripts will be expressed in a simple, compact, interpreted language.

Our programming model follows that of the proxies of [Shapiro86]: any access to an object is through a local copy of the manager code. Although this may sound a little extreme at first sight, we also allow the local manager code to be but a shadow of the real thing—for example, a stub (perhaps with some local caching, perhaps not), that just forwards all requests to a remote copy of the full manager code.

4 DataMesh phase 1

DataMesh phase 1 is concerned with the Jungle *device* layer. Although this might be thought of as having little to do with file systems *per se*, we have learned that there are a great many advantages to considering the designs of both the file system and the device itself simultaneously.

The basic goal of the DataMesh phase 1 work is to develop high performance, flexible block-level servers. The interface to these servers is at the level of read and write operations on fixed-size blocks (chunks) of data. Initially, we will adhere to the existing SCSI command set so that the devices we are developing can be added directly to an existing system; in later work we plan to make minor extensions to it (such as the provision of a free command, and a trial implementation of the SCSI-3 disk-based file system proposal) to enable some of our more aggressive ideas to be tested out.

Our approach is straightforward and direct: we are constructing a prototype hardware/software testbed (the DataMesh *preprototype*); gathering and synthesizing a wide range of workloads; and using these to evaluate a range of different new disk system designs.

4.1 Phase 1 infrastructure

The DataMesh preprototype is constructed from an 8-node transputer system manufactured by Parsytec, running the Helios operating system from Perihelion. Each node has an Inmos T800 transputer, 4MB RAM and a SCSI controller chip. Seven of the nodes have an attached 5.25" SCSI disk drive; the eighth has a SCSI connection to a host workstation. The whole assemblage is attached to an HP C(an IBM PC clone) through which access to the building LAN is provided. We are able to

1. UNIX is a registered trademark of Unix System Laboratories in the USA and other countries.

cross-compile software on our workstations, download it into the testbed, and debug it—all without leaving our offices.

On the testbed we have already developed a SCSI target-mode device driver so that we can send requests to it from the workstation as if it were a disk drive; a trial implementation of a parallel RAID 5 disk array; and a measurement and performance monitoring harness.

We have been gathering and using a number of real-system workload traces to drive our simulations. The HP-UX operating system supports an interface known as the measurement system, which can capture system events with 1µs timer resolution. Typical events include file system calls, file buffer cache misses (and hits), and disk device requests—enqueues at the device driver, and physical I/O start and completion. This is a powerful tool to use for testing the effects of device designs on file system performance. Our traces cover several months worth of activity on a single-user workstation and a local timesharing system; we are currently extending both the breadth and detail of our trace collection.

We have also put together a number of tools for synthesizing workloads given descriptions of patterns of accesses [Wright92].

4.2 Some of our results so far

Modern disks have different properties than ones studied intensively by file system designers in the 1970s. For example, it now takes only 1.3 revolutions of the disk for the head to seek from the outside edge to the inside, which suggests that existing request-scheduling algorithms (such as the SCAN algorithm used by UNIX [Coffman72]) that order requests by seek distance are no longer optimal. By taking advantage of rotation position information we have developed a family of algorithms that provide much better throughput while also avoiding starvation effects. About double the throughput can be obtained for a constant response time on real traces [Jacobson91, Seltzer90]. We are continuing this work in collaboration with Giorgio Gallo at the University of Pisa.

While the previous technique improves throughput significantly under high loads, it does little to help improve latency (other than to reduce queueing delays). A separate idea, published as [English92] improves write latencies. We called the approach *Loge*: it uses an indirection table in the disk to map logical block addresses to physical ones. In combination with around 3–5% of the disk reserved as free blocks (in much the same way that the 4.2BSD file system [McKusick84] reserves 10% of the disk to improve layout performance), write latencies for small (4KB) writes can be roughly halved, with minimal effect on read performance; Loge can also sustain roughly half the raw bandwidth of the disk for random small writes. The trick is to write to the nearest available free block: by doing so, seek and rotational latencies can almost be eliminated for small transfers. The benefits are obviously relatively greatest for the smallest transfers, especially those involved in synchronous writes, such as forcing data or the tail of a log file to disk for a commit. (Such writes can of course be avoided by file system restructuring: LFS [Rosenblum91] delays writes—thereby sacrificing availability—to achieve larger asynchronous transfers; the clustering techniques of [McVoy91] improve large sequential transfers, but don't improve random I/O.)

Although the above two techniques are best applied in the disk itself because they rely on fast access to rotation-position information, the next idea could be applied in the file system as well—possibly with even better results. (Indeed, some similar work has been done in this area on whole-file layout [Staelin91].) The presence of the Loge indirection table gives us additional benefits beyond the fast writes: we can now consider optimizing the layout of the disks on a per-block basis for better read performance, for example, by shuffling the most actively read data to the center of the disk. An investigation of techniques to do this [Ruemmler91] suggested that performance improvements of up to 20% can be achieved on an *already optimized* 4.2BSD file system layout, and should be able to do much better with the additional randomness introduced by Loge. The ability to do per-block reorginzation was crucial to the performance gains: larger units showed much less impresive gains. An important point about this approach is that repetitive sequences of random I/O can be well handled by this technique, while most other work merely improves sequential I/O. We believe this technology could usefully be combined with LFS-like techniques to improve the cleaning function and arrange for data that is accessed together to be physically co-located. We are continuing this work in collaboration with Dave Musser at Rensselaer Polytechnic Institute. One

thing we learned is that—even in the presence of large file system caches—there is a surprisingly high locality in disk accesses.

Finally, we have investigated the effects of replicating data dynamically to achieve better read performance. (We did this work on whole files, unlike the rest of the results reported here.) The basic idea is to make multiple copies of frequently-accessed data—our experiments used multiple disks, but this could also be done on a single disk—and select between the copies at read time for lowest-latency accesses [Lo90]. Straightforward replication can increase update times (since multiple copies all have to be written to), so we experimented with different policies for making and discarding replicas dynamically. Our results showed that (a) dynamic replication can provide up to about 25% reduction in average file system disk access time; (b) keeping only one copy on an update is roughly twice as good as keeping them all; (c) within these parameters, all the policies that we tested worked about equally well. We believe these incremental benefits should also be obtainable from more aggressive file system implementations, too.

4.3 Current research

Current and near-term work in the DataMesh project is concentrating on phase 1, the block level server. Our pursuits cover a number of areas including:

- Unidisk virtual devices: fast-write disks (Loge), disk request scheduling, disk shuffling.
- Multidisk virtual devices: MultiLoge: Loge across multiple spindles.
- SCSI extensions: sparse addresses, multiple spigots, tagged data, load balancing across devices, and a SCSI-3 file system prototype.
- High availability: decentralized RAID 5, with no single point of failure and distributed parity calculation and error recovery; strong recovery guarantees combined with high performance (Mime [Chao92]).

We are currently bringing up the multiple-policy framework on the DataMesh 1 preprototype hardware, and are implementing the Loge and Mime devices for it, as well as simple disk and RAID disk array devices for comparison purposes.

5 Conclusions

The work described here represents a radical departure in the design of file systems: it considers the entire chain of functionality from record-level operations down to and including the device controller functionality. Our work to date has exposed several possibilities for greatly improved performance and functionality. Doubtless there will be more.

References

[Chao92] Chia Chao, Robert English, David Jacobson, Alex Stepanov and John Wilkes. *Mime: a high performance storage device with strong recovery guarantees.* CSP technical report HPL–CSP–92–9, Hewlett-Packard Laboratories, March 1992.

[Coffman72] E. G. Coffman, L.Klimko and B.Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal on Computing* 1(3):269–79, September 1972.

[English92] Bob English and Alex Stepanov. Loge—a self-organizing disk controller. *Proceedings of Winter USENIX'92* (San Francisco, CA) Jan. 1992.

[Jacobson91] David Jacobson and John Wilkes. *Disk scheduling algorithms based on rotation position.* CSP technical report HPL–CSP–91–7,Hewlett-Packard Laboratories, Feb. 1991.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, August 1984.

[McVoy91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 33–43, 21–25 January 1991.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review* **25**(5):1–15, 13 October 1991.

[Ruemmler91] Chris Ruemmler and John Wilkes. *Disk shuffling.* Technical report HPL–91–156, Hewlett-Packard Laboratories, Oct. 1991.

[Seltzer90] Margo Seltzer, Peter Chen and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX* (Washington, DC), pp. 22–26 January 1990.

[Shapiro86] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proceedings of 6th International Conference on Distributed Computing Systems* (Cambridge, Mass), pp. 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[Staelin91] Carl Staelin and Hector Garcia-Molina. Smart filesystems. In *Proceedings of the Winter 1991 USENIX* (Dallas, TX), pp. 45–51, 21–25 January 1991.

[Wilkes89] John Wilkes. DataMesh — scope and objectives: a commentary. DSD technical report HPL–DSD–89–44, Hewlett-Packard Laboratories, July 1989.

[Wilkes91] John Wilkes. DataMesh — parallel storage systems for the 1990s. *Proceedings of the 11th IEEE Mass Storage Symposium* (Monterey, CA), October 1991.

[Wilkes91a] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review* **25**(1):56–9, January 1991.

[Wright92] Rebecca Wright, A library for synthetic multithread trace generation. CSP technical report HPL–CSP–92–1, Hewlett-Packard Laboratories, January 1992.

Zebra: A Striped Network File System

John H. Hartman
John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720
jhh@sprite.Berkeley.EDU
ouster@sprite.Berkeley.EDU

Abstract

This paper presents the design of Zebra, a striped network file system. Zebra applies ideas from log-structured file system (LFS) and RAID research to network file systems, resulting in a network file system that has scalable performance, uses its servers efficiently even when its applications are using small files, and provides high availability. Zebra stripes file data across multiple servers, so that the file transfer rate is not limited by the performance of a single server. High availability is achieved by maintaining parity information for the file system. If a server fails its contents can be reconstructed using the contents of the remaining servers and the parity information. Zebra differs from existing striped file systems in the way it stripes file data: Zebra does not stripe on a per-file basis; instead it stripes the stream of bytes written by each client. Clients write to the servers in units called *stripe fragments*, which are analogous to segments in an LFS. Stripe fragments contain file blocks that were written recently, without regard to which file they belong. This method of striping has numerous advantages over per-file striping, including increased server efficiency, efficient parity computation, and elimination of parity update.

1 Introduction

Zebra is a network file system architecture designed to provide both high performance and high availability. This is accomplished by incorporating ideas from log-structured file systems, such as Sprite LFS [Rosenblum91], and redundant arrays of inexpensive disks (RAID) [Patterson88] into a network file system. From log-structured file systems Zebra borrows the idea that small, independent writes to the storage subsystem can be batched together into large sequential writes, thus improving the storage subsystem's write performance. RAID research has focused on using striping and parity to obtain high performance and high availability from arrays of relatively low-performance disks. Zebra uses striping and parity as well, resulting in a network file system that stripes data across multiple storage servers, uses parity to provide high availability, and transfers file data between the clients and the storage servers in large units. The notable features of Zebra can be characterized as follows:

Scalable performance. A file in Zebra may be striped across several storage servers, allowing its contents to be transferred in parallel. Thus the aggregate file transfer bandwidth can exceed the bandwidth capabilities of a single server.

High server efficiency. Storage servers are most efficient handling large data transfers because small transfers have high overheads. Large transfers are relatively simple to achieve for large files, but small

This work was supported in part by the National Science Foundation under grant CCR-8900029, the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

files pose a problem. Client file caches are effective at reducing server accesses for small file reads, but they aren't as effective at filtering out small file writes [Baker91]. Zebra clients use the storage servers efficiently by writing to them in large transfers, even if their applications are writing small files.

High availability[1]. Zebra can tolerate the loss of any single machine in the system, including a storage server. Zebra makes file data highly available by maintaining the parity of the file system contents. If a server crashes its contents can be reconstructed using the parity information. The use of parity allows Zebra to provide the availability of a system that maintains redundant copies of its files while requiring only a fraction of the storage overhead.

Uniform server loads. File striping causes the load incurred by a heavily used (*hot*) file to be shared by all of the storage servers that store the file. In a traditional network file system a hot file only affects the performance of the server that stores it, requiring that hot files be carefully distributed among all of the servers to balance the load.

Zebra is currently only a paper design, although a prototype is being implemented in the Sprite operating system [Ousterhout88]. This paper describes the design of Zebra, not the prototype implementation. The rest of this paper is organized as follows. Section 2 discusses striping and its application to a network file system, Section 3 discusses the use of parity to provide high availability, Section 4 gives an overview of the Zebra architecture, and Section 5 describes the Zebra design in more detail. Section 6 covers Zebra's status and future work, and Section 7 is a conclusion.

2 Why Stripe?

Traditional network file systems confine each file to a single file server. Unfortunately this means that the rate at which a file can be transferred between the server and a client is limited by the performance characteristics of that one server, such as its CPU power, its memory bandwidth and the performance of its I/O controllers. This makes it difficult to improve the performance of the network file system without improving or replacing the server. Striping a file over several servers allows those servers to transfer the file in parallel, so that the aggregate transfer rate can be much higher than that of any one server. The file transfer performance of the file system can be improved simply by adding more servers.

A striped network file system has several economic advantages over a traditional network file system. First, the storage servers do not need to be high-performance, nor do they need to be constructed out of special-purpose hardware. Servers in a traditional network file system are often among the more expensive and high-performance machines in the system. In contrast, storage servers in a striped network file system can be relatively modest machines, thereby improving their cost/performance and reducing the fraction of the total system cost that they represent. Second, a striped network file system allows clients to be upgraded without requiring server upgrades as well. The increased client performance can be matched by increasing the number of servers, rather than replacing them. Both of these considerations make a striped network file system an economically attractive alternative to a traditional network file system.

The idea of using striping to improve data transfer bandwidth is not a new one. It's often used to improve the performance of disk subsystems by striping data across multiple disks attached to the same computer. Mainframes and supercomputers have used striped disks for quite a while [Johnson84]. The term *disk striping* was first defined by Salem and Garcia-Molina in 1986 [Salem86]. More recently there has been lots of interest in arrays of many small disks, originating with the paper by Patterson et al. in 1988 [Patterson88]. All of this work focuses on aggregating several relatively slow disks to create a single logical disk with a much higher data rate.

In recent years striping has been applied to file systems as a whole. In these file systems the blocks of each file are striped across multiple storage devices. These storage devices may be disks, as in HPFS [Poston88], I/O nodes in a parallel computer, as in CFS [Pierce89] and Bridge [Dibble90], or they may be network file servers as in Swift [Cabrera91]. It is important to note that these systems stripe on a per-file basis,

1. The distinction between availability and reliability, while it is important, is not particularly relevant to this paper. The arguments made here regarding the availability of Zebra also apply to its reliability.

therefore they work best with large files. Small files are a kind of Catch-22: if they span all the storage devices then the amount stored on each device will be small, causing the devices to be used inefficiently, but if small files aren't striped then the system performance when writing small files will be no better than that of a non-striped system. Applications that write many small files will not see a performance improvement.

Striping also serves as a load-balancing mechanism for the storage devices. Ideally the storage devices would have identical loads, so that no one device saturates and becomes a bottleneck. If files are constrained to a single storage device then care must be taken to ensure that hot files are evenly distributed across the devices. Striping eliminates the need for careful file placement by distributing files over multiple devices. The load caused by a heavily used file is shared by all the devices that store it, thus reducing the variance in device loads.

3 Availability

Striping can potentially reduce the availability of a network file system, since each file in the system is distributed over several storage servers. The loss of any one of these servers will cause the file to be unavailable. If a striped file system with multiple servers replaces a file system with a single server, then the availability of the system will be reduced (assuming the servers in both systems have the same failure rate).

Network file systems often improve availability by maintaining redundant copies of each file. Redundant copies are advantageous because they allow the system to withstand server failures (provided at least one copy remains available), and they can allow the system to tolerate network partitions. If each section of the partitioned network contains a copy of each file then files can continue to be accessed without interruption. Redundant copies do have disadvantages, however. Additional storage is needed for the extra copies, and there are complexities and overheads involved in keeping the copies up-to-date.

An alternative approach is to maintain error-correcting information that allows missing data to be reconstructed. RAID systems favor this scheme. For example, a simple parity scheme will allow the system to tolerate the loss of a single server. One block of data from all but one server is XOR'ed together to produce a parity block which is stored on the remaining server. Should one of the blocks of data become unavailable it can easily be recomputed from the other blocks of data and the parity block. The advantage of this approach is that the storage required for the parity blocks is much less than is needed for redundant copies. Swift proposes to use parity to tolerate server failures for just this reason.

It is easier to implement a parity mechanism for a RAID storage system than for a network file system, however. A RAID is usually connected to a host computer, through which all data transfers to and from the array must pass. This makes the host a convenient location to compute parity. A network file system doesn't have a comparable centralized location, however. This makes it more difficult to compute parity across physical storage blocks or file blocks. If files are written randomly, or written by multiple clients simultaneously, then no single location may contain all of the data needed to perform the parity calculation. File updates are also problematic, since they require updating the parity of the modified blocks as well. The new parity must be computed from the old and new contents of the block, potentially causing several server accesses per update. In addition, the update of a block and its parity must be an atomic operation, since data could be lost if the two get out of sync. Ensuring that two writes to two different servers are atomic is likely to be complex and expensive.

4 Zebra Stripes

Zebra differs from existing striped network file systems in that it does not stripe on a per-file basis. Instead it stripes on a per-client basis: all of the new data from a single client is formed into a single logical stream, regardless of which files the data belongs to. This stream is then striped across the storage servers. The data written to the servers in a single pass by a single client is called a *stripe*. The portion of a stripe that is written to each server is called a *stripe fragment*. Clients compute the parity of the stripe fragments as they are written. At the end of a stripe the client writes out the resulting parity fragment and begins a new parity computation. Note that each stripe is written by a single client. Multiple clients may be writing to the storage

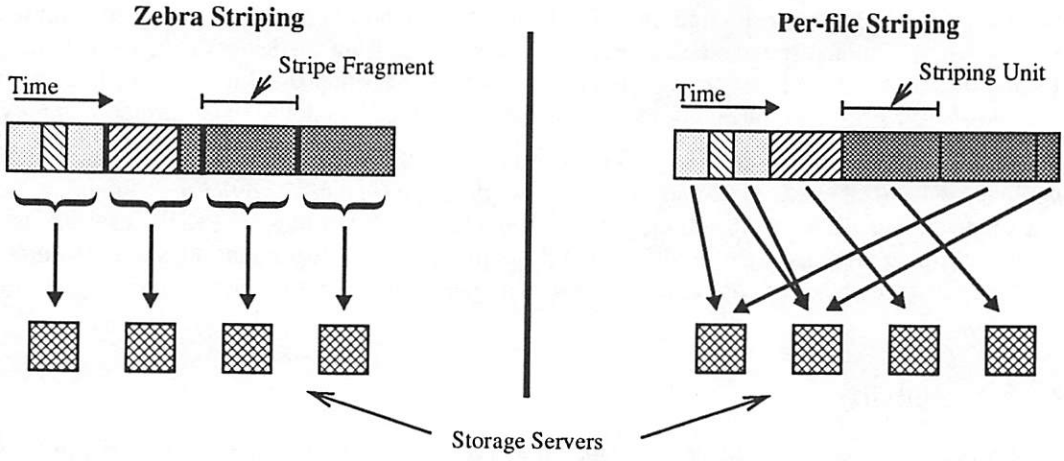


Figure 1: Zebra striping vs. per-file striping. This figure shows the same sequence of file data being written in both Zebra and in per-file striping. Each shaded region represents a piece of data written to a single file. Regions with the same shading belong to the same file. Zebra clients pack together file blocks and write them to the storage servers in large transfers called *stripe fragments*. Per-file striping requires more transfers, because small writes aren't batched. In this example the striping unit (the maximum amount of data written to a single server) in the per-file system is the same size as a stripe fragment in Zebra. Parity writes are not shown.

servers simultaneously, but they are all writing to distinct stripes. Figure 1 illustrates the Zebra striping mechanism.

Zebra, like all file systems, must maintain *metadata* that records the physical storage location for each file block. When a file block is accessed the file's metadata is used to determine which storage location to use. In Zebra the *file manager* is responsible for managing the metadata (See Section 5.2 for details). Once a client has written a stripe it sends a summary of the stripe's contents to the file manager, so that the metadata can be updated accordingly.

Clients are responsible for reconstructing files from their constituent stripe fragments. Upon opening a file for reading the client obtains the metadata for the file from the file manager. To read the file the client uses the metadata to determine which stripe fragments to access, then retrieves the desired portions of those fragments from the storage servers and reassembles them into the file.

Stripes in Zebra are analogous to segments in a log-structured file system. They are large conglomerations of file data that can be efficiently transferred. The data stored in a stripe exhibits temporal locality, rather than spatial locality, i.e. a stripe contains blocks that were written during the same period of time rather than blocks from the same file. Like segments, stripes are immutable objects. Once they are written they cannot be modified. Free space is reclaimed from stripes by *cleaning* them (see Section 5.3).

Zebra's method of striping has several advantages over per-file striping. First, the striping algorithm is relatively simple. No special mechanisms are needed for handling small files, randomly written files, or file updates; their bytes are simply merged into the client's stream of bytes. Large files that are sequentially written will be striped in a manner similar to per-file striping, however. Each file will be divided into stripe fragments and striped across the servers. Second, parity computation and management is simplified. Parity is easily computed because each client computes the parity of the stripes it produces. The parity computation doesn't require any coordination between clients, or additional data transfers between the servers and the clients. Since stripes are immutable and parity is never updated, Zebra also avoids having to atomically update a file and its parity. A simple timestamp mechanism is sufficient for ensuring that a stripe and its parity are consistent. If a client should crash while in the process of writing a stripe the timestamps are used to determine how many of the stripe fragments were actually written prior to the crash, so that the stripe's parity can be updated accordingly.

Striping the logical stream of bytes from each client, rather than files, improves the performance of writing small files. Clients write stripe fragments to the storage servers, not files or file blocks. This decouples the size of the files used by the client applications from the size of the accesses made to the storage servers. Applications that write many small files will see a performance improvement over traditional network file systems because the files will be batched together and written to the server in large transfers.

5 Zebra Design

Figure 2 illustrates the components of Zebra. The storage servers store file data, in the form of stripe fragments. The file manager manages the file name space and keeps track of where files are located. The stripe manager handles storage management by reclaiming the space occupied by data that is no longer used. The rest of this section describes these components in more detail.

5.1 Zebra Storage Servers

The Zebra storage servers are merely repositories for stripe fragments. They create new fragments in response to client requests and retain the fragments until they are deleted by the stripe manager. The fragments are opaque to the storage servers -- the servers know nothing about the files or blocks that the fragments contain. This simple functionality makes it possible to implement the storage server in a variety of different ways. One option is to build a special-purpose storage server that has been optimized for storing stripe fragments. Another is to store stripe fragments as files in a local file system. This approach is not only easy to implement but it also allows the storage servers to be traditional network file servers, some of whose files happen to be Zebra stripe fragments.

The simple functionality of the storage servers is well-suited to machines that emphasize I/O capabilities rather than CPU speed. For example, the RAID-II project at Berkeley [Lee92] is building a storage system that has a high-bandwidth connection between its disk array and the network. Unfortunately, it is relatively expensive for the host CPU to access the data that passes over that connection. Traditional network file systems are likely to run slowly on such an architecture since the host must process each file block. A Zebra storage server, on the other hand, only performs per-fragment processing and is better able to take advantage of this type of architecture.

5.2 Zebra File Manager

The Zebra file manager manages the file system metadata, i.e. the file system name space and the mapping of logical file blocks into stripe fragments. Clients send all name space modifications, such as file creation and deletion, as well as file open and close events to the file manager. This allows the file manager to ensure the consistency of the metadata even if clients are modifying it concurrently. The file manager is a

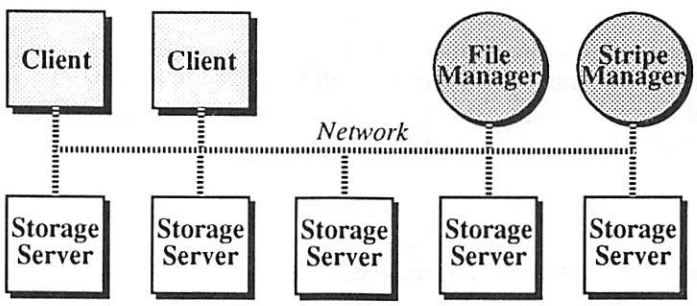


Figure 2: Zebra schematic. Squares represent individual machines; circles represent logical entities. The file manager and the stripe manager can run on any machine in the system, although it is likely that one machine will be designated to run both of them.

critical resource; the file system cannot be accessed if its metadata is unavailable. Zebra employs a backup file manager that can take the place of the primary file manager should the primary fail. During normal operation the primary file manager logs all changes in the metadata to the backup. If the primary should fail the backup uses this information to reconstruct the current state of the metadata.

The modification rate of the metadata in Zebra is likely to be higher than in traditional file systems. This is because the mapping of a file block to a stripe fragment changes when the file block is modified. Clients determine the storage location for a file block by packing it into a stripe fragment and writing the fragment to the next server in its rotation. If a client overwrites an existing file block then the new version of the block will probably be stored in a different fragment. When a client writes a stripe it must tell the file manager which file blocks it contains so the file manager can update its metadata.

The centralization of name service and file mapping information on the file manager is a potential performance bottleneck. One technique for eliminating this bottleneck is to have the clients cache both name and mapping information. Client name caching has been shown to be effective at reducing the load on the name server in a network file system [Shirriff92]. By caching whole directories of file names and their mapping information the Zebra clients can eliminate the need for contacting the file manager each time they modify the name space or access a file.

5.3 Stripe Manager

The Zebra stripe manager is responsible for managing the storage space on the storage servers by reclaiming free space from existing stripes. Its function is similar to the *segment cleaner* in a log-structured file system. The manager keeps a list of all stripes in the system, and which file blocks they contain. As blocks are deleted or overwritten the list is updated accordingly. When free space is needed a stripe is *cleaned*[2], a process in which its live data is read and then written to a new stripe. The storage servers are then notified that the space currently allocated to the cleaned stripe can be reused. The stripe manager represents a centralized location in which the live data from stripes that are cleaned can be formed into stripe fragments and their parity computed. Zebra uses a backup stripe manager to ensure that the stripe manager is always available.

Cleaning's impact on system performance is proportional to the amount of live data in the stripes that are chosen to be cleaned. Ideally the stripes would not contain any live data, so that cleaning them would not cause any data transfers. Measurements of Sprite LFS show that the write traffic to the disk due to cleaning is relatively low (for non-swap file systems it is between 2% and 7% of the overall write traffic to the disk) [Rosenblum91]. Further research is required to determine if Zebra exhibits the same behavior.

6 Status and Future Work

Zebra is currently a paper design. Implementation of a prototype began in the spring of 1992, and should be completed by late 1992. Once the prototype is completed it will be measured and compared to existing network file systems in the following ways:

- Performance under a variety of workloads, each of which has a different distribution of file sizes and read/write ratios. The emphasis will be on Zebra's performance running workloads with lots of large files (supercomputer workload), and workloads with lots of small files (UNIX workload). The workloads will probably be synthetic.

- Parity's cost, in terms of CPU cycles, network bandwidth, and storage server resources.

- Stripe cleaning's impact on performance.

- Tolerance of failures of the storage servers, the file manager, the stripe manager, and the clients.

---

2. The algorithm for choosing which stripe to clean is beyond the scope of this paper.

7 Conclusion

Zebra applies ideas from log-structured file system research and RAID research to network file systems, resulting in a system that has the following advantages over existing network file systems:

Scalable performance. The transfer rate for a single file is proportional to the number of servers in the system.

Cost effective servers. Zebra servers do not need to be high-performance machines or have special-purpose hardware. The performance of the file system can be increased by adding more servers, rather than replacing the existing ones.

High server efficiency. Server overhead is reduced because clients write to the storage servers in large transfers, and the servers do not interpret the contents of the stripe fragments they store. There are no per-file or per-block server overheads associated with writing a stripe fragment to a storage server.

Simple parity mechanism. Parity is computed by the clients as they write out stripe fragments. Parity is never updated, so expensive parity update computations and atomic operations are not needed.

Uniform server loads. Striping reduces the variation in server load by distributing hot files across multiple storage servers.

The Zebra architecture promises to provide high-performance file access to both large and small files. Large files are striped to improve their transfer rate; small file writes are batched together to reduce server overheads. The result is a cost-effective, scalable, highly-available network file system that can provide file service to a supercomputer as easily as to a workstation.

8 Acknowledgments

Tom Anderson provided helpful comments on drafts of this paper. Alan Smith suggested the name "Zebra".

9 References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff and John K. Ousterhout, "Measurements of a Distributed File System", *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP)*, Asilomar, CA, October 1991, 198-212.

[Cabrera91] Luis-Felipe Cabrera and Darrell D. E. Long, "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates", *Computing Systems 4*, 4 (Fall 1991), 405-436.

[Dibble90] Peter C. Dibble, "A Parallel Interleaved File System", Ph.D. Thesis, University of Rochester, 1990

[Johnson84] O. G. Johnson, "Three-dimensional wave equation computations on vector computers", *Proceedings of the IEEE 72* (January 1984).

[Lee92] Edward K. Lee, Peter M. Chen, John H. Hartman, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson and David A. Patterson, "RAID-II: A Scalable Storage Architecture for High Bandwidth Network File Service", Technical Report UCB/CSD 92/672, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, February 1992.

[Ousterhout88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, "The Sprite Network Operating System", *IEEE Computer 21*, 2 (February 1988), 23-36

[Patterson88] David A. Patterson, Garth Gibson and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, Chicago, IL, June 1988, 109-116.

[Pierce89] Paul Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem", *Proceedings of the Fourth Conference on Hypercubes*, Monterey CA, March 1989.

[Poston88] Alan Poston "A High Performance File System for UNIX", NASA NAS document, June 1988

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP)*, Asilomar, CA, October 1991, 1-15.

[Salem86] Kenneth Salem and Hector Garcia-Molina, "Disk Striping", *Proceedings of the 2nd International Conference on Data Engineering*, February 1986, 336-342.

[Shirriff92] Ken Shirriff and John Ousterhout, "A Trace-driven Analysis of Name and Attribute Caching in a Distributed File System", *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, CA, January 1992, 315- 331.

Optimal Write Batch Size in Log-Structured File Systems *

Scott Carson and Sanjeev Setia

Department of Computer Science
University of Maryland
College Park, MD 20742
{sdc,setia}@cs.umd.edu

Abstract

Log-structured file systems create a performance benefit by maximizing the rate at which disk write operations can occur. However, this benefit comes at the expense of individual read operations, which can experience long queueing delays due to batched write service. This paper presents an analytic derivation of the write batch size that minimizes read response time, while retaining the benefit of write-batching. Simulations that relax the assumptions of the analytic model demonstrate that the analytic result can be used to advantage in a practical setting.

1 Introduction

Recent studies [RO92] have shown that log-structured file systems can exhibit substantial system performance gains over "traditional" file systems by batching write operations. On the other hand, in a recent paper [CS92], we showed that bulk write operations can have an adverse effect on read performance experienced by processes. The conflict between these results is due to their different performance metrics; in the former, disk bandwidth utilization is considered important, while in the latter, response time for synchronous operations is of greater interest. In this paper, we continue to hold that response time is the more important metric, while acknowledging that write-batching can benefit both system and process performance. Based on this assumption, we derive the write batch size that minimizes read response time. Most significantly, we show that the optimal write batch size is largely insensitive to system load, and that it is primarily dependent on disk characteristics. This means that it is possible to create a nearly optimal write-batching policy based on statically determined parameters.

Log-structured file systems [RO92, OD89] are based on the idea that write operations can be postponed, in a disk cache, until there is a large backlog. The write operations are then executed as a small number of disk operations, each of which transfers a large amount of data. This amortizes a small number of seek and rotational latencies over a large number of data blocks, and allows the log-structured file system to write data at a rate that approaches the transfer rate of the disk, given a sufficiently high write load.

Unfortunately, the performance benefits of this method do not extend directly to read operations, since in general, they are not part of the batching scheme. Our analysis of

*The work described in this paper was supported in part by a grant from the Digital Equipment Corporation. The views expressed herein are those of the authors.

the "periodic update" write policy used in some operating systems showed that creating large sequences of write operations that are served in bulk produces an interruption of read service that causes read operations to experience long queueing delays. In our study, we only considered groups of write operations that were served singly, without the benefit of write-batching. Although write-batching does ameliorate the problem by reducing the collective service time of a group of writes, it is clear that the same phenomenon can occur in write-batched systems, albeit to a lesser degree.

2 Solution

In non-batching, write-bulk-arrival systems, one simple way to reduce read queueing delays is to give read operations non-preemptive priority over writes. This eliminates the interruption of service caused by the write bulk. In write-batching systems, however, no such scheme is possible, since the writes are serviced as a small number of large disk operations (each a "segment" in LFS). One solution is to break up each batched write operation into a series of smaller batch operations, and to give read operations non-preemptive priority over these write batches. This introduces a fundamental conflict between write batch size and read response time.

Suppose that a read request arrives at the disk queue, and that a series of write requests (each of which represents a batch) is already in queue. Minimizing the size of these write batches (and increasing the number of batches accordingly) then minimizes the waiting time for that read operation, since the read must only wait for the batch write operation in service to complete. If a read request arrives when no writes are in queue, then it is serviced immediately. Maximizing the size of these write batches (and reducing their number correspondingly) minimizes the fraction of time the disk is busy servicing writes, and therefore minimizes the probability that an arriving read will find the system busy. This suggests that there is an optimal write batch size that balances the ability of reads to "slip in" between write operations with the need to complete write operations quickly. The derivation of this optimal write batch size is the focus of this paper.

Our strategy for finding the optimal write batch size is as follows. First, we express the read response time as a function of write batch size (assuming read priority) analytically, under simplifying assumptions. Next, we derive the optimal write batch size, and show that it is dependent only on disk characteristics: seek time, rotational latency, and transfer rate. Recognizing that the assumptions used in the analytic model may not hold in practice, we then relax the assumptions and simulate the system. Our simulations show that the optimal write batch size predicted by the analytic model is a reasonable estimate of the optimal write batch size in a realistic system. Thus, the results of the analytic model can be used to drive a disk service policy in a nearly-optimal way. Next, we discuss the implementation of this scheme, and show example calculations based on our model and on the characteristics of several popular disks.

3 Analytic Model

This section presents an analytic model of response time for read operations for the disk service scheme described in the previous section. Under this scheme, read operations are given non-preemptive priority over write operations, and the write batch size is a policy

parameter. In concept, the disk server maintains separate queues for read and write requests (see Figure 1).

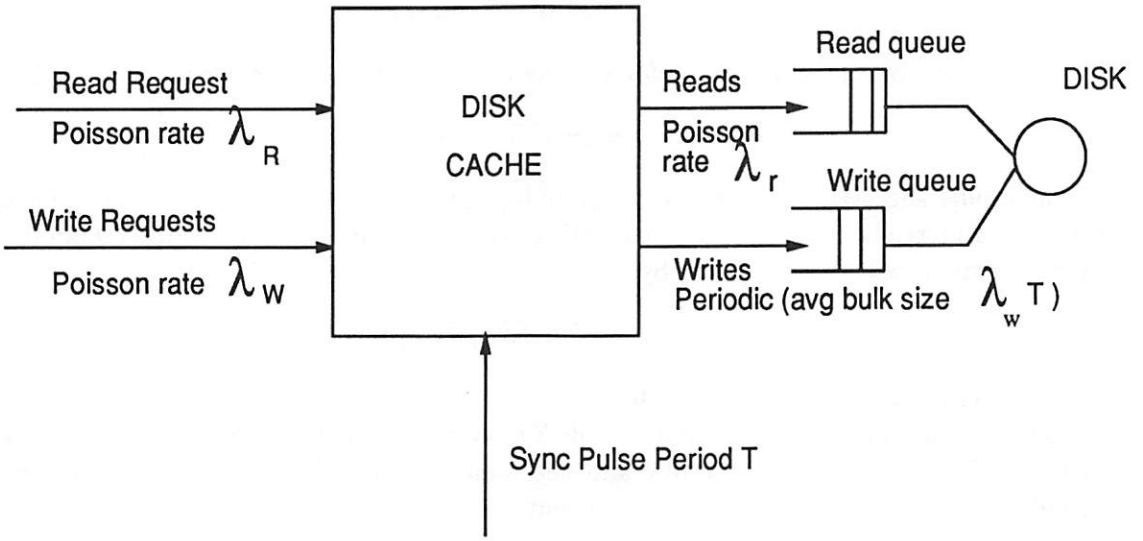


Figure 1: The Model

We assume, for now, that read operations arrive at the cache according to a Poisson process with a rate of λ_R requests/s. In the model, these requests enjoy a fixed cache hit-ratio of h_r, $0 \leq h_r \leq 1$ so that the resulting disk read traffic is a Poisson process with arrival rate $\lambda_r = (1 - h_r)\lambda_R$.

A typical method of batching writes is to save them in a main memory cache and flush them on a periodic basis, in response to a "sync pulse". LFS [RO92] is a "pure" periodic update policy, since all write operations are delayed and written out at periodic intervals. We assume that writes arrive at the cache according to a Poisson process with rate λ_W. Let T be the length of the interval between two updates. The write hit-ratio $h_w(T), 0 \leq h_w(T) \leq 1$ is the fraction of writes that are to blocks that are already dirty or that will not be written to disk due to deletion. At the beginning of each update period, the dirty blocks are written to disk. Under LFS, the dirty blocks in the cache are aggregated into "segments" of a fixed size, and sent to the disk queue; under our modified scheme, these segments are further divided into batches of size c blocks. The number of dirty blocks written at the update interval is Poisson distributed with mean $\overline{A} = \lambda_w T = (1 - h_w(T))\lambda_W T$. For a batch size of c blocks, $\lfloor \frac{\overline{A}}{c} \rfloor$ full batches and (at most) one partial batch are placed in the write disk queue. To simplify our analysis, we assume that the number of blocks written out every T second is *always* $\overline{A}$; thus the partial batch has $p = \overline{A} \bmod c$ blocks.

The disk service time for writing out a batch (or a partial batch) is a function of the batch size. The service time, $s(c)$ can be written as

$$s(c) = sk + rot + tr(c)$$

where sk is the seek time, rot is the rotational latency and $tr(c)$ is the transfer time. The transfer time is a function of the batch size, c, as well as the characteristics of the disk, while the seek time and rotational latency depend only upon disk characteristics. In general, the individual random variables that compose the service time are not independent. However, for the purpose of this analysis we shall assume that seek time and rotational latency are

independent, and have stationary probability distributions. Then, the average service time for writing out a batch of size c is given by

$$\overline{s}_w = \overline{sk} + \overline{rot} + \overline{tr}(c) \tag{1}$$

Similarly, the average service time for writing out a partial batch containing p blocks is

$$\overline{s}_p = \overline{sk} + \overline{rot} + \overline{tr}(p)$$

In our model successive read requests are independent. In reality, this may not always be the case and reads may benefit from various optimizations. However, in our model the average read service time is given by

$$\overline{s}_r = \overline{sk} + \overline{rot} + \overline{tr}(1)$$

where $\overline{tr}(1)$ is transfer time for one block.

The variance of the service time for a disk operation is given by the sum of the variances of the seek time, rotational latency and the transfer time. In general, the transfer time distribution is deterministic. We assume that

$$\overline{tr}(c) = k \cdot c \tag{2}$$

where k is a constant that depends upon the disk. Thus the service time variance for an operation is the sum of the seek time and rotational latency variances. That is, for a disk operation, o,where $o = w, p$ or r denoting a full batch write, a partial batch write or a read respectively, we have

$$\sigma_o^2 = \sigma_{sk}^2 + \sigma_{rot}^2$$

Note that the variance in the service time for disk reads, full batch writes and partial batch writes is the same. Henceforth we shall drop the subscript o in σ_o^2. Then, the coefficient of variation of service time for a disk operation, o is given by

$$C_{s_o}^2 = \frac{\sigma^2}{\overline{s}_o{}^2} \tag{3}$$

Thus we can see from the expression above, that increasing the batch size, c, results in a reduction in the coefficient of variation of the write service time.

The system shown in Figure 1 can be considered to be a head-of-line priority queueing system with three classes of customers: reads, full batch writes and partial batch writes, with reads having priority over the other two classes of customers.

The average response time for reads can be readily derived from results in [Kle76] for a k-class priority queueing system:

$$\overline{R} = \overline{s}_r + \frac{\rho_r \overline{s}_r (1 + C_{s_r}^2) + \rho_w \overline{s}_w (1 + C_{s_w}^2) + \rho_p \overline{s}_p (1 + C_{s_p}^2)}{2(1 - \rho_r)} \tag{4}$$

where

$$\rho_w = \lfloor \frac{\overline{A}}{c} \rfloor \cdot \frac{\overline{s}_w}{T}$$

and

$$\rho_p = \begin{cases} \frac{\overline{s}_p}{T} & \text{if } \overline{A} \bmod c > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that the read response time depends upon the write load, but is not dependent upon the manner in which the writes are presented to disk. Thus, the analysis applies equally to any scheme for aggregating writes, and is not restricted to the "periodic update" policy. Further note that the write load is divided into two components: the write load due to full batch writes (ρ_w) and the write load due to partial batch writes (ρ_p).

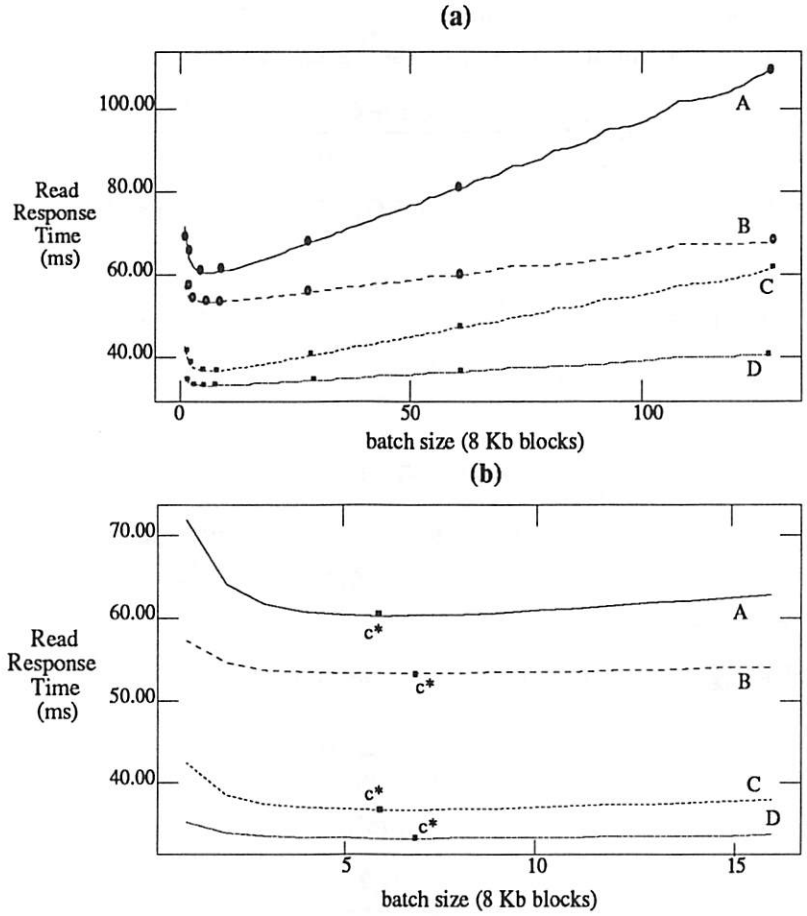


Figure 2: (a) Read Response time (in ms) vs batch size (in 8 Kb blocks) assuming Poisson read arrivals. A: $\rho_r = 0.6, \varrho_w = 0.6$; B: $\rho_r = 0.6, \varrho_w = 0.2$; C: $\rho_r = 0.2, \varrho_w = 0.6$; D: $\rho_r = 0.2, \varrho_w = 0.2$. Curves are from the analytical model; marked points are simulation results. (b) shows the same curves blown up for batch sizes less than 16.

To validate the model, a number of simulations were run; all simulation results reported are with 95% confidence intervals of 1% about the mean. For the simulations, we considered a hypothetical disk with 815 cylinders, in which the seek time function $sk(d) = 1.775 + 0.725d^{0.5}$ ms, where d is the seek distance in cylinders. This results in a seek time of 2.5 ms for a single-cylinder seek and 22.5 ms for an end-to-end seek. We assume that the rotational latency is uniformly distributed between 0 and 16 ms. The block size is 8 kilobytes, and the disk transfers data at 2 Mbyte/s, so that the transfer time is given by $tr(c) = 4c$ ms, i.e. the transfer time for one block is 4 ms. In Figure 2(a), we compare the results of our model with simulations. As the figure shows our model predicts the read response time quite accurately. Note that the simulations and the model share all assumptions about read and write arrival patterns and service times. However, in the simulation, gathered statistics

are based on read and write arrivals, not the mean number of reads and writes as used in the model.

As Figure 2(b) shows, the response time appears to have a minimum for a certain batch size, irrespective of the read and write loads. [1] We now explore this observation analytically.

The batch size, c, can only take on integral values. Hence, equation (4) for the read response time is only meaningful if c is an integer. However, for the moment, let us assume that c is a real number, and that there are no partial batch writes. Then equation (4) can be rewritten as a continuous function of the batch size, c:

$$\overline{R}(c) = \overline{s}_r + \frac{\rho_r \overline{s}_r (1 + C_{s_r}^2) + \rho_w \overline{s}_w (1 + C_{s_w}^2)}{2(1 - \rho_r)} \tag{5}$$

Substituting $\rho_w = \frac{\lambda_w \overline{s}_w}{c}$ and for $\overline{s}_w$ and $C_{s_w}^2$ from equations (1) and (3) respectively, we have

$$\begin{aligned}
\overline{R}(c) &= \overline{s}_r + \frac{\rho_r \overline{s}_r (1 + C_{s_r}^2)}{2(1 - \rho_r)} + \frac{\lambda_w}{2(1 - \rho_r)c}[\overline{s}_w^2 + \sigma^2] && (6) \\
&= \overline{s}_r + \frac{\rho_r \overline{s}_r (1 + C_{s_r}^2)}{2(1 - \rho_r)} + \\
&\quad \frac{\lambda_w}{2(1 - \rho_r)c}[(\overline{sk} + \overline{rot} + k \cdot c)^2 + \sigma^2] && (7) \\
&= \overline{s}_r + \frac{\rho_r \overline{s}_r (1 + C_{s_r}^2)}{2(1 - \rho_r)} + \frac{\lambda_w}{2(1 - \rho_r)}[\frac{D^2 + \sigma^2}{c} + k^2 c + 2Dk] && (8)
\end{aligned}$$

where $D = \overline{sk} + \overline{rot}$.

Differentiating $\overline{R}(c)$ with respect to the batch size, c, we have

$$\frac{\partial \overline{R}(c)}{\partial c} = \frac{\lambda_w}{2(1 - \rho_r)}[k^2 - \frac{D^2 + \sigma^2}{c^2}] \tag{9}$$

Setting the derivative equal to 0, we obtain the optimal batch size c^*

$$c^* = \left(\frac{D^2 + \sigma^2}{k^2}\right)^{0.5} \tag{10}$$

What is remarkable about the result above is that the batch size that minimizes the read response time is *independent* of the read and write loads and only depends upon the characteristics of the disk. For the simulations plotted in Figure 2, $\sigma^2 = 44.2305$, $k = 4$ blocks/ms and $D = 23.75$ leading to $c^* = 6.17$. However, as the batch size can only take on integral values, we have $c^* = 7$ or $c^* = 6$. It can be seen in Figure 2(b), that the read response time is minimum for a batch size of 6 or 7 for all combinations of read and write loads.

4 Relaxing the Assumptions

The model presented in the previous section has several simplifying assumptions that allow it to be analyzed. If the results obtained through the model are to prove useful, we must

[1] Note that the disk utilization due to write batches is a function of the batch size while the disk utilization due to reads is independent of the batch size. We define the *offered load* due to writes as $\varrho_w = \lambda_w \cdot \overline{s}$, where $\overline{s}$ is the service time for writing a single block.

be sure that the results will also be true for real systems. In this section, we relax several assumptions made in the previous section and examine the their impact on the results of the last section. Relaxing the assumptions makes our model intractable and thus the results in this section are obtained only through simulation.

All simulations reported in this section were obtained using a simple event-driven simulator. Read operations were generated according to either an exponential or a hyperexponential interarrival time distribution. Writes were generated similarly, but were "saved up" for 30 seconds, at which point they were "flushed" to the disk queue. Disk service times were generated by adding a uniformly distributed, random rotational latency, the seek time corresponding to a uniformly distributed, random "next cylinder," and the transfer time corresponding to either a single block (for reads) or a batch-sized block (for writes). Disk characteristics were those described in the previous section. Simulations were run long enough to provide 95% confidence intervals of 1% about the mean.

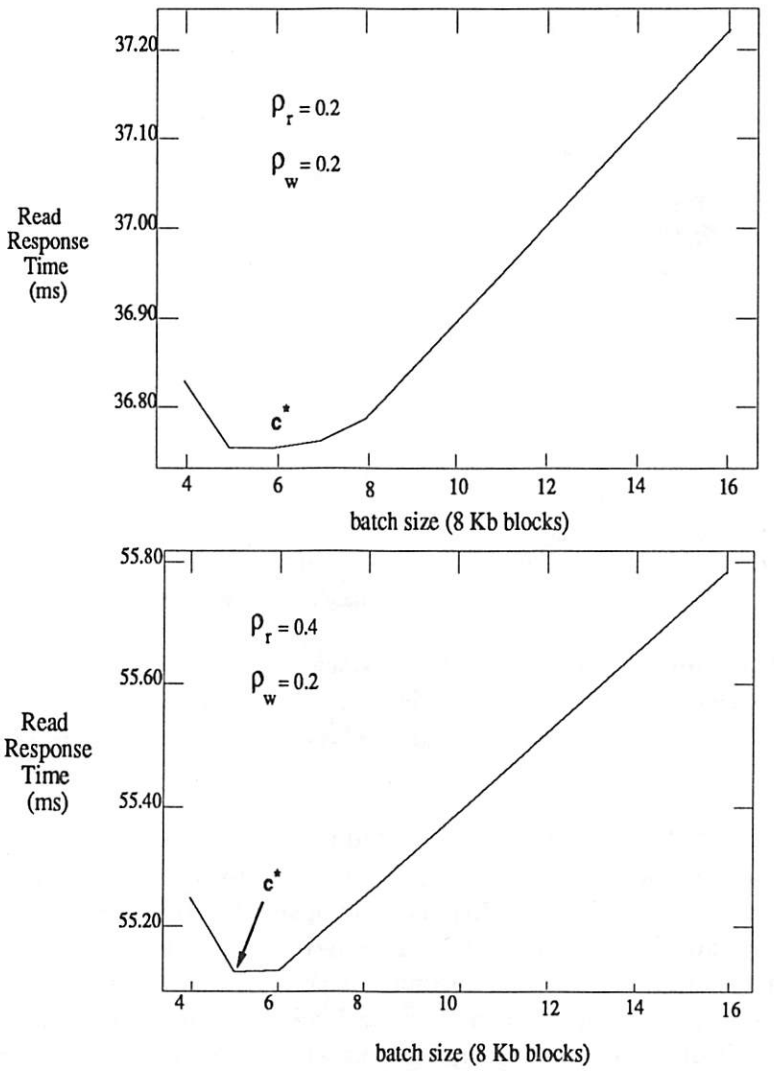


Figure 3: Read Response time (im ms) vs batch size (in 8 Kb blocks) for hyper-exponential read arrivals with $C_a^2 = 5$. Note that the batch size can only have integral values. Curves are linearly interpolated between batch size values.

The first assumption we address is that of Poisson read and write arrivals to the disk cache. This leads to Poisson read arrivals to the disk, and to the number of blocks being written out to the disk at the sync pulse being Poisson distributed. Most experimental studies of I/O traffic for UNIX-like filesystems have observed that both read and write traffic is quite bursty [OdCH+85, BHK+91]. In order to model this burstiness, in our simulations we now assume that both reads and writes arrive according to a hyperexponential process, which has a coefficient of variation (C_a^2) > 1. This has two effects: first, reads arrive in a much more bursty fashion (depending upon the coefficient of variation), and second, there is a larger variance in the number of blocks written to the disk at each sync pulse.

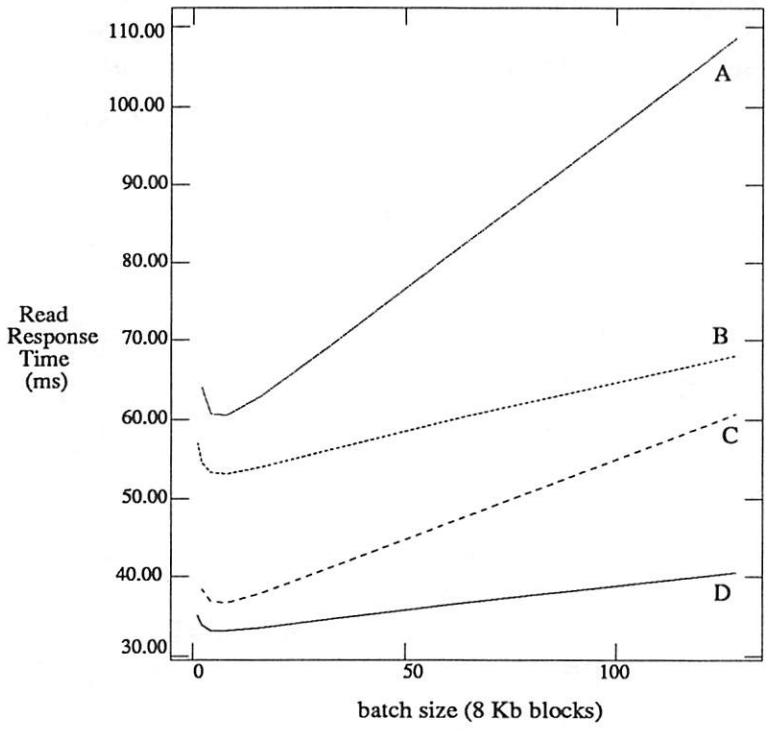


Figure 4: Read Response time (in ms) vs batch size (in 8 Kb blocks) for hyperexponential write arrivals (with $C_a^2 = 5$) to the cache. A: $\rho_r = 0.6, \varrho_w = 0.6$; B: $\rho_r = 0.6, \varrho_w = 0.2$; C: $\rho_r = 0.2, \varrho_w = 0.6$; D: $\rho_r = 0.2, \varrho_w = 0.2$. Curves are linearly interpolated between batch size values.

The results of the simulations are plotted in Figure 3. As the graph shows, the optimal batch size for various read and write load combinations is no longer the same. Thus the result derived in the previous section that the optimal batch size is independent of the load is not true for non-Poisson read and write arrivals. However, what is noteworthy about the plots in Figure 3 is that the optimal batch size is *still strongly dependent* upon disk characteristics. For all the read and write load combinations in the figure, the optimal batch size is within 2 blocks of the optimal batch size for the model in the last section. The effect of the bursty read arrivals is to push the optimum towards a smaller batch size. The effect of the bursty writes is less pronounced as Figure 4 shows (compare with Figure 2). This is understandable because the writes are aggregated by the cache and only delivered to the disk at the beginning of the update interval. These results show that the optimal batch

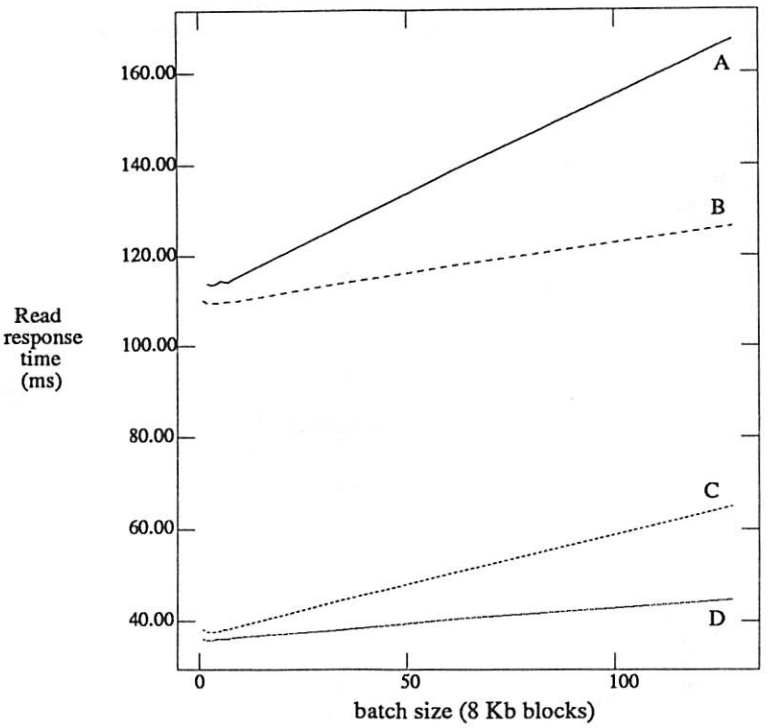


Figure 5: Read Response time (in ms) vs batch size (in 8 Kb blocks) taking "zero-seeks" into account. A: $\rho_r = 0.6, \varrho_w = 0.6$; B: $\rho_r = 0.6, \varrho_w = 0.2$; C: $\rho_r = 0.2, \varrho_w = 0.6$; D: $\rho_r = 0.2, \varrho_w = 0.2$.Curves are linearly interpolated between batch size values.

size obtained from the model in the previous section can be used by file system designers to get an idea about the optimal batch size for real systems.

We now relax another assumption of our model. In the model and in the simulation described above, the disk service time for writing a batch is the sum of the seek time, rotational latency and transfer time. This assumes that in writing out a batch, one always has to pay the price of a seek. However, in practice, this is only true when a read request(s) arrives during the time a batch is being written out. Since the read has higher priority the read request(s) will be serviced next, thus necessitating a seek to get back to write the next batch of the segment being written. However, if no read request arrives while the write batch is written, the next batch can be written out without a seek. This is because the head will already be positioned on the right track. Thus only a rotational latency and transfer time cost must be paid for writing the next batch.

The results of the simulation that take this effect into account are shown in Figures 5 and 6. We observe that the average read response time is not affected substantially. However, the optimal batch size has been pushed even further to the left than in Figure 3. This is understandable because the effect of the "zero-seeks" is to reduce the value of the constant D (as the average seek distance is reduced), thus leading to a reduction in the optimal batch size. In Figure 6, the optimal batch size appears to be 2 or 3 blocks, depending on the read and write loads.

Finally, we consider another optimization that is possible in real systems. If we fix the batch size such that the batch fits exactly on a track or on multiple tracks, then while

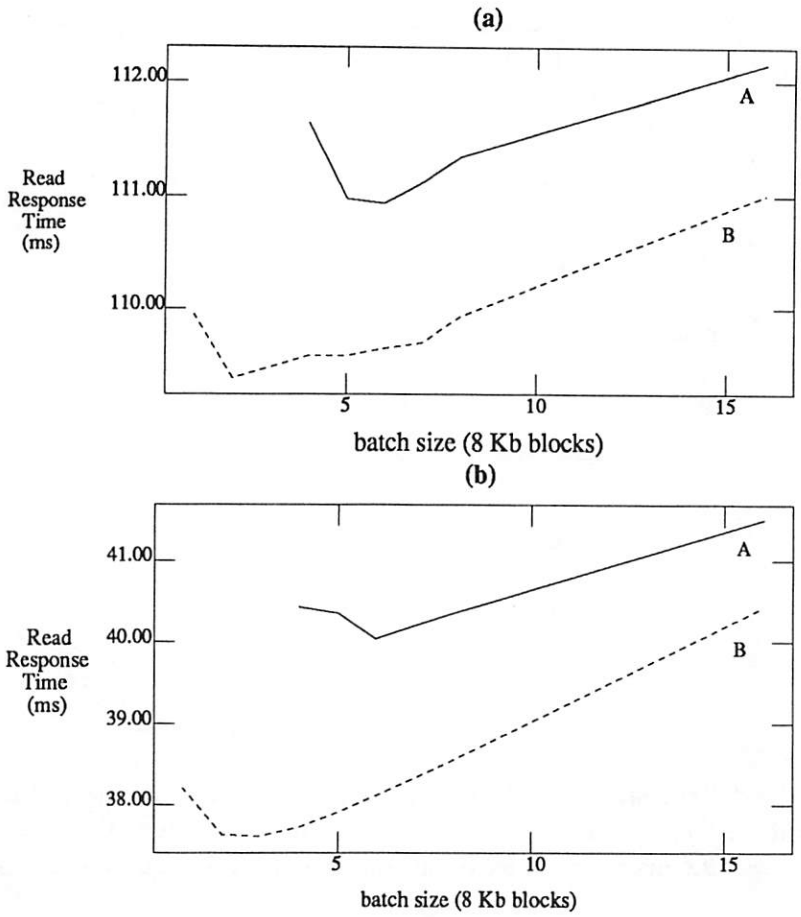


Figure 6: Read Response time (in ms) vs batch size (in 8 Kb blocks) taking "zero-seeks" into account for (a) $\rho_r = 0.6, \varrho_w = 0.2$ and (b) $\rho_r = 0.2, \varrho_w = 0.6$. Curves are linearly interpolated between batch size values. A : hyperexponential read arrivals ($C_a^2 = 5$); B: hyperexponential read arrivals and "zero-seeks".

writing out a batch of the segment, one no longer has to pay a cost for rotational latency. This is because as soon as the disk head is positioned upon the track to be written it can initiate the disk transfer. Thus in this case, one only has to pay the cost of seek (if it all) and data transfer.

Simulation results for this optimization are shown in Figure 7. As expected, the simulations show that read response time is reduced for batch sizes that are multiples of 4 8-Kbyte blocks, but is otherwise unaffected. With this optimization in place, the optimal batch size may be a multiple of the track size, as it is in this case.

5 Discussion

The results of this study can be used to create a parameterized service policy for disk systems that provides the advantages of write-batching, while (nearly) minimizing average response time for read operations. In this section we consider several practical issues: use of the results, implementation, and example policy parameters.

We considered several realistic situations: bursty read and write arrivals, taking advan-

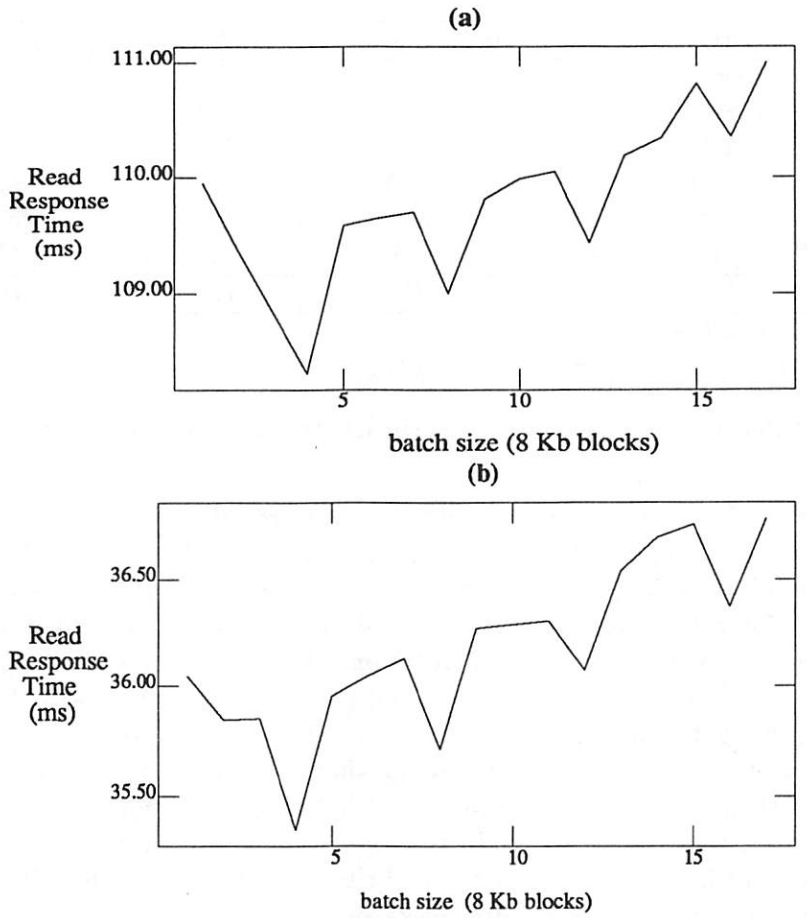


Figure 7: Read Response time (in ms) vs batch size (in 8 Kb blocks) assuming track-sized batches for (a)$\rho_r = 0.6, \varrho_w = 0.2$ and (b) $\rho_r = 0.2$, $\varrho_w = 0.2$. Curves are linearly interpolated between batch size values.

tage of disk head position when servicing writes, and writing complete tracks at a time. In each case, these conditions introduced some deviation from the analytic model. In particular, the actual response times were substantially different from those predicted by the model. However, the optimal batch size was similar to (smaller than) the predicted value in each case. This suggests that it is possible to derive general policy guidelines from the model.

The write-batching policy described in this paper is easy to implement. The policy is driven by a single parameter, c^*, that describes the optimal batch size. The log-structured file system does not need to know about this policy; instead, it is implemented in the disk driver itself. Suppose that a log-structured file system delivers groups of write operations to the disk queue. These groups of write operations can be of arbitrary size. The disk driver then repeatedly issues batches of writes, each batch containing n sectors, to the disk. However, between each write batch, the driver checks for the presence of read requests in the disk queue, and services all reads before servicing the next write batch.

Since the nearly-optimal value of n is fixed for a given disk, it is possible for the disk driver to use a table of batch sizes, indexed by disk type, to govern the policy. Table 1 shows c^* for three popular one-gigabyte disks.

Drive Model	IBM 0663-H12	HP 97549	Fujitsu M2266S/H
Size ($Gbytes$)	1	1	1
Mean Seek (ms)	14.4	24.9	21.1
Seek Var. (ms^2)	18.9	61.6	39.6
Mean Rot. (ms)	6.95	7.47	8.3
Rot. Var. (ms^2)	16.1	18.6	23.2
Transfer Rate (Mb/s)	2.4	2.2	2.5
c^* ($Kbytes$)	52	72	74
Realized Write Transfer Rate (Mb/s)	1.2	1.1	1.3

Table 1: Optimal Batch Sizes (in Kbytes) for some contemporary disks

The values for seek time mean and variance were calculated from manufacturer's specifications of minimum (track-to-track) and maximum (end-to-end) seek times, assuming a uniform, random seek-distance distribution and constant head acceleration/deceleration. The values for rotational latency mean and variance were calculated based on the disk rotation speed, assuming a uniform rotational latency distribution. Transfer rates were calculated based on disk rotation speed and track density. The optimal cluster size, $c*$, is reported in kilobytes rather than 8-kilobyte blocks as in the previous sections.

Although these disks are similar enough that they might be considered interchangeable in practice, their characteristics produce distinct effects that agree with intuition about the optimal write cluster size. The IBM disk, for example, seeks and rotates more quickly than the others. This leads to a smaller optimal cluster size, since the penalty for breaking up a larger cluster is smaller. The HP disk seeks and transfers more slowly than the Fujitsu disk, but since it rotates more quickly, the penalty for breaking up a cluster is again smaller, leading to a smaller optimal cluster size. Finally, the Fujitsu disk transfers data more quickly than the others, so more data per disk operation must be transferred to make the seek and rotational penalties insignificant.

In practice, it can be expected that the optimal cluster size will be slightly smaller than c^*, since c^* is calculated without the effects of seek optimization, track-caching, and bursty arrivals. Although more investigation is needed before firm policy guidelines can be presented, the results of the previous section show that the deviation from c^* under realistic conditions will be small. However, it must also be noted that the larger the batch size, the more write load the system can sustain. Thus, it may be appropriate to ignore those effects that reduce the optimal batch size, and to use c^* instead. In addition, for systems that can take advantage of full-track writes, the best choice for the write batch size may be a multiple of the track size.

Another measure shown in Table 1 is the realized write transfer rate. This is calculated as the ratio of c^* to the sum of seek, rotational, and transfer delays. Since the realized write transfer rate is substantially smaller than the maximum transfer rate provided by the disk, systems that require sustained, high write rates may require a larger batch size than c^*.

As disk technology evolves, the optimal write batch size may change. Although it is difficult to predict the exact nature of future optimizations, it can be expected that disks will become increasingly dense, and will offer higher transfer rates. If seek and rotational latencies remain similar to their current values, then this means that c^* will increase in the future. Likewise, synchronized multi-disk systems, designed for high transfer rates, will

require larger write batches than single-disk systems of similar technology.

6 Conclusions

The results of this paper demonstrate that if read response time is an important performance metric, care must be taken in the design of write-batching file systems. Specifically, making the write cluster size too large penalizes reads by making them wait for long write operations, while making the write cluster size too small penalizes reads by wasting disk bandwidth. The optimal cluster size is dependent on load and access patterns, but is dominated by a function of disk characteristics.

Although our results are preliminary, in that they have not been confirmed by measurement on a real system, they do suggest that it is possible to devise a batching policy that works well with statically-determined parameters. Further, our results contrast sharply with some of the basic design objectives of log-structured file systems, such as maximizing the size of write batches to increase throughput. Although there are cases where write throughput is the most important performance criterion, in most general-purpose environments read response time is more important. Our results indicate that in such cases, batch sizes should be considerably smaller than they would be otherwise.

References

[BHK+91] M. Baker, J. Hartman, M. Kupfer, K. Shirrif, and J. Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th Symp. on Operating System Principles*, 1991.

[CS92] S. D. Carson and S. K. Setia. Analysis of the Periodic Update Policy for Disk Cache. *IEEE Transactions on Software Engg.*, January 1992.

[Kle76] L. Kleinrock. *Queueing Theory II: Applications*, volume 2. J. Wiley and Sons, 1976.

[OD89] J. Ousterhout and F. Douglis. Beating the I/O Bottleneck: The Case for Log-Structured File Systems. *Operating Systems Review*, January 1989.

[OdCH+85] J. Ousterhout, H. da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the Unix 4.3 BSD File System. In *Proc. of 10th Symp. on Operating System Principles*, 1985.

[RO92] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM TOCS*, February 1992.

A Recovery Protocol for Spritely NFS

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory
250 University Avenue
Palo Alto, California 94301
mogul@decwrl.dec.com

Abstract

NFS suffers from its lack of an explicit cache-consistency protocol. The Spritely NFS experiment, which grafted Sprite's cache-consistency protocol onto NFS, showed that this could improve NFS performance and consistency, but failed to address the issue of server crash recovery. Several crash recovery mechanisms have been implemented for use with network file systems, but most of these are too complex to fit easily into the NFS design. I propose a simple recovery protocol that requires almost no client-side support, and guarantees consistent behavior even if the network is partitioned. This proves that one need not endure a stateless protocol for the sake of a simple implementation.

I also tidy up some loose ends that were not addressed in the original experiment, but which must be dealt with in a real system.

1. Introduction

NFS has been extremely successful, in large part because it is so simple and easily implemented. The NFS "stateless server" dogma makes implementation easy because the server need not maintain any (non-file) state between RPCs, and so need not recover state after a crash.

Statelessness is not inherently good. Since many NFS operations are non-idempotent and might be retried due to a communication failure, to get reasonable performance and "better correctness" the server must cache the results of recent transactions [Jusz89]. Such cache state is not normally recovered after a crash, although this exposes the client to a possible idempotency failure.

A more serious problem with NFS statelessness is that it forces a tradeoff between inter-client cache consistency and client file-write performance. In order to avoid inconsistencies visible to client applications, NFS client implementations (by tradition, rather than specification) force any delayed writes to the server when a file is closed. This ensures that when clients use the following sequence:

Writer	**Reader**
open()	
write()	
close()	
	open()
	read()
	close()

the reader will see the most recent data, *if* the writer and reader explicitly synchronize so that the reader's *open* takes place after the writer's *close*.

Unfortunately, this means that the *close* operation is synchronous with the server's disk. Since most files are small [Bake91b, Oust85], this means that most file writes effectively become

synchronous with the server's disk, and NFS clients spend much of their time waiting for disk writes to complete. Also, although many files have very short lifetimes and are never shared, and need never leave the client's cache, NFS forces them to the server's disk and so wastes a lot of effort. Finally, NFS does not guarantee any form of cache consistency for simultaneous write-sharing, with the result that occasional consistency errors plague NFS users.

The Sprite file system [Nels88] solves these problems by introducing an explicit cache-consistency protocol. The fundamental observation is that write-sharing is rare, and can be detected by the server if clients report file opens and closes (not done in NFS), so the write-through-on-close policy can be eliminated. Instead, when write-sharing does occur, Sprite turns off all client caching for the affected file, and thus provides true consistency between client hosts.

2. Goals and design philosophy

Before describing the changes I propose for Spritely NFS, I want to briefly list the goals that I think such a system should meet:

- **Simplicity**: Spritely NFS was a successful experiment partly because it required minimal changes to an NFS implementation, and almost no changes to any other code. Any improved version should avoid unnecessary complexity, especially on the client side.
- **Consistency**: Spritely NFS should provide guaranteed cache consistency at all times. A partial guarantee is no improvement on NFS, since an application cannot make use of a partially-guaranteed property.
- **Performance**: Spritely NFS is not worth doing unless its performance, even with recovery, is better than that of NFS. While Spritely NFS also promises better consistency, that in itself would not convince many users to switch.
- **Reliability**: Spritely NFS should be no less reliable than NFS or the local Unix file system. (Note that I am satisfied with matching the lesser of these reliabilities in a given situation; NFS is sometimes, but not always, more reliable than a local Unix file system, and Spritely NFS sometimes must give up these NFS properties.)
- **No-brainer operation**: System managers should not need to do anything special to manage a Spritely NFS system. In particular, they should not need to adjust parameter values.
- **Incremental adoption**: Spritely NFS clients should interoperate with NFS servers, and vice versa. Otherwise, users will not have much of an incentive to adopt Spritely NFS, since this would mean replacing large parts of their infrastructure all at once.

3. Review of Spritely NFS

Spritely NFS [Srin89] was an experiment to show that a Sprite-like consistency protocol could be grafted onto NFS, and to show that the performance advantage of Sprite over NFS was in large part due to the consistency mechanism rather than other differences between Sprite and Unix®. In this section I will summarize the design of Spritely NFS.

Two new client-to-server RPC calls are added to the basic NFS suite, *open* and *close*. The *open* call includes a mode argument that tells the server whether the client is writing the file or just reading it.

The NFS server is augmented with a ''state table,'' recording the consistency state of each currently-open file. In Spritely NFS, this state table is relevant only to the *open* and *close* RPCs; all other client RPCs are handled exactly as in NFS. When a client issues an *open* RPC, the server makes an entry in its state table and then decides, based on other state table information, if the specified open-mode conflicts with uses by other clients. If the *open* is conflict free, the server (via the RPC return value) notifies the client that it can cache the file. Otherwise, the client is not allowed to cache the file.

In some cases, a conflict may arise after a client has opened a file and has been allowed to cache it. For example, the first client host might open a file for write, and be allowed to cache it, and then a second host might open the same file. At this point, in order to maintain consistency, the first client must stop caching the file.

For this reason, Spritely NFS adds server-to-client "callback" RPCs to the NFS protocol. When a server decides that a client must stop caching a file, it does a callback to inform the client. A client with cached dirty blocks may have to write these blocks back to the server before replying to the callback RPC.

Spritely NFS clients need not write-through dirty blocks when a file is closed. The server keeps track of closed-dirty files and can ask the client to write the blocks back if another client opens the file for reading, but otherwise the writer client can write the blocks back at its own leisure. A client with closed-dirty blocks might even remove the file before the blocks are written back, thus avoiding wasted effort.

4. Loose ends in Spritely NFS

Besides lacking support for recovery, Spritely NFS failed to address a few other issues that need to be solved in a real system. I will propose solutions for these before describing the recovery protocol, since some of these issues complicate the basic recovery mechanism.

4.1. Dealing with ENOSPC

One problem with the delayed-write-after-close policy is that one or more of these writes might fail. In NFS, since the client implementation forces all writes to the server before responding to the *close* system call, an application which checks the return value from both *write* and *close* calls will always know of any write failures. Not so in Spritely NFS, since the failure might occur long after the application has released its descriptor for the file (or even after the application has exited). This could cause trouble for applications that do not explicitly flush their data to disk.

There are three categories of error that can occur on a client-to-server *write* operation:

1. **Communications failure**: the network is partitioned or the server crashes, and the RPC times out before the failure is repaired.
2. **Server disk hardware error**: the disk write operation fails, or the disk fails after the write completes.
3. **Server out of disk space**: no space is available on the server disk.

The first error can be turned into a delay by simply retrying the RPC until the server responds[1]. If the client crashes in the interim, then the dirty block is lost ... but this is no different from a normal local-filesystem delayed write in Unix.

The second error is not generally solvable, even by a strict write-through policy. It is true that the NFS approach will report detectable write failures, but these are increasingly rare (because techniques such as bad-block replacement can mask them). Again, normal Unix local-filesystem semantics does not prevent this kind of error from occurring long after a file has been closed.

The third error (ENOSPC, in Unix terms) is the tricky one. We want to report these to the application, because it might want to recover from the condition, and because there is no obvious

[1]This is not true if the client uses a "soft mount," which turns RPC timeouts into errors rather than retries. Soft mounts are generally thought of as living dangerously, although delaying writes after a *close* does make them even more dangerous. Perhaps soft-writes-after-close should be made "harder" as long as the client has enough buffer cache to avoid interference with other operations.

way for the underlying file system mechanism to recover from ENOSPC. (Also, unlike the other two kinds of errors, one cannot avoid ENOSPC errors through fault-tolerance techniques.)

My understanding is that Sprite does not completely solve this problem; that is, Sprite applications can believe their writes are safe but the delayed writes pile up in a volatile cache because the server is out of space. I would be curious to know if and how AFS deals with this (although in AFS, the client cache is "stable" and so the consequences are less urgent).

I propose solving the ENOSPC problem by changing the Spritely NFS *close* RPC to reserve disk space for the remaining dirty blocks. That is, when a dirty file is closed, the client counts up the number of dirty bytes and requests that the server reserve that much disk space for the file. The server may respond with an ENOSPC error at this point, in which case the client can revert to a write-through on close policy (note that we will allow the server to respond to *close* with ENOSPC even when enough space does exist, so the client should attempt the writes and report an error to the application only if a write actually fails.).

If a client, after it has obtained a reservation, decides it does not need to write back some of the dirty blocks (because it instead removes or truncates the file), it can issue a *release* RPC to release part or all of a reservation. Note that a *write* RPC might not actually change the size of a file, so once the last dirty block is gone from the client, the client must set the reservation to zero with a *release* RPC[2] (*release* might be implemented as *close* with a reservation request equal to zero.)

When a server receives a reservation request, attached to a *close* RPC, it should arrange with the underlying disk file system to reserve some of the remaining free space for the file in question. (If the space is not available, the server responds to the *close* with ENOSPC.) The file system need not actually allocate space on disk for the reservation; rather, it only needs to keep a count of the number of free bytes and the number of reserved bytes, and ensure that the difference never becomes negative.

The server keeps track of a file's reservation in its state table entry. When a server handles a *write* RPC for a closed file, it decrements the reservation, and does a special kind of write to the underlying file system that tells the file system to decrement the count of reserved bytes[3]. If a client in the closed-dirty state tries to write more blocks than its reservation allows, the writes will fail (this is to prevent cheating and inconsistencies in the reservation count). Note that a client which has not asked for a reservation can write blocks as long as non-reserved space exists in the file system. Note also that the underlying file system must prevent local applications from allocating disk space if the non-reserved space drops to zero.

If a client crashes while holding a reservation, or simply never makes use of it, the space could be tied up indefinitely. Thus, the server should set a time limit on any reservation grant (perhaps in proportion to the number of blocks reserved; if a client reserves space for a billion bytes, it is unlikely that they could all be written back within a short interval. A server might also refuse to honor a reservation for more than a few second's worth of disk writes). When the time limit expires and space is low, the server can reclaim the reservation by doing a callback (to force the client to write back the dirty blocks).

---

[2]The original Spritely NFS design does not have an explicit operation to cause a transition from the closed-dirty state to the closed state. This could cause the server's state table to fill up. In the original system, the plan was that the server would do a callback to force the transition, if necessary. The *release* RPC solves this problem.

[3]The count is decremented by the amount of space actually allocated in the file system, not the size of the write. This avoids a double decrement of the reservation when a *write* RPC is retried.

A client that fails to respond to the callback, perhaps because of a network partition, might end up being unable to write dirty blocks if the server reclaims its reservation. Since a partition might last arbitrarily long, there is not much that can be done about this: conceptually, this is the same as a disk failure. However, to avoid provoking this problem, a server might refrain from reclaiming timed-out reservations as long as sufficient free space remains.

One subtle problem can occur with this scheme if two processes on one client are writing the same file. After one successfully closes the file (i.e., the server grants a reservation), if the other client extends the file so much that the server runs out of disk space, some part of the file might not be written to the server. This is not an entirely contrived example; the file might be a "log," appended to by multiple processes. The client implementation can preserve correct semantics in such a case by ordering the disk writes so that none of the blocks dirtied after the *close* are written to the server before the other dirty blocks of that file.

4.2. Directory caching

Spritely NFS did not address the issue of directory caching. This is important because a large fraction of NFS traffic consists of directory lookups and listings. Many NFS implementations cache directory entries, but because NFS has no consistency protocol these caches must time out quickly and can nevertheless become inconsistent.

Recent measurements on Sprite suggests that it is better to cache (and invalidate) entire directories rather than individual entries, since a directory is often the region of exploitable locality of reference [Shir92]. This nicely matches the Spritely NFS model; the client simply does an *open* on a directory before doing *readdir* RPCs, and keeps the result of the *readdir* in a cache. When the client removes a directory from its cache, it does a *close* RPC to inform the server. If another client modifies the directory (using an RPC such as *create*, *remove*, *rename*, etc.), then the server does a callback to cause the first client to invalidate its cache.

One open issue is whether a client should write-through any changes (i.e., creations, renames, or removals), or if directory changes can be done using write-behind. My hunch is that the latter is far more complex, especially because it makes it much harder to provide the failure-atomicity guarantee that Unix has traditionally attempted for directory operations. If only write-through is allowed, then *open* on a directory always allows the client to cache; it serves solely to inform the server of which clients might need callbacks when an entry is changed.

In order to avoid the potential for thrashing that would result if a large shared directory was continually being modified by several clients, when a directory is invalidated (via a callback) the client should remember this and not re-cache the directory for a period of several minutes[4]. That is, when an invalidate occurs, the client should *close* the directory (so that the server will not need to do further callbacks) and not cache the result of *readdir* during the inhibition period.

There are a few issues that arise which prevent a client from simply replacing a bunch of *lookup* RPCs with one *readdir*:

- **Lack of attributes and filehandle information**: The NFS *lookup* RPC returns a filehandle and file attributes value that is not returned by *readdir*.
- **Lack of symbolic link values**: If the directory entry is a symbolic link, the client must do a *readlink* to resolve the reference.
- **Extra delay**: When a client wants to lookup one entry in a large directory, it should not have to wait for the server to provide the entire directory contents before continuing.

Together, these suggest the following approach, assuming a cold cache:

---

[4]This is one of those parameters that I wanted to avoid, but it can be set fairly long because during this period the directory will simply revert to the normal uncached NFS behavior. Traces of real directory activity might help here.

1. The client starts by doing the *lookup* RPC that it would normally do.
2. In parallel, it performs an *open* RPC and then a *readdir* RPC (or a series of *readdir* RPCs, if the directory is large) to load the directory into the cache.
3. The result of the *lookup* is inserted into the cache, in such a way that it can be invalidated if the server does a callback on this directory.
4. If the *lookup* indicates that the file is a symbolic link, the client does a *readlink* and inserts that into the cache.

Once the directory is opened, the client can cache the results of *lookup* and *readlink* RPCs, since if anyone else modifies the directory the client will receive a callback.

One would think that doing the *readdir* RPC is unnecessary, since the information returned by *readdir* does not obviate the need to do *lookup* (and perhaps *readlink*). This is not entirely true; the reason is that applications (especially the shells) often attempt to lookup files that do not exist. Without the full set of names in the directory cache, the client cannot avoid going to the server to *lookup* these non-existent names. The consistency guarantee provided by Spritely NFS ensures that such "negative" caching is accurate, and so many *lookup* RPCs can be avoided.

4.3. Caching attributes for unopened files

Spritely NFS provided consistency for file attributes (length, protection, modification time, etc.) only for open files. Clients, however, often use the attributes of files that they won't (or can't) open; commands such as `make`, `ls -l`, or `du` fall into this category. Because such commands are so frequent, NFS implementations are forced to provide "attribute caching" using a probabilistic consistency mechanism: cached attributes time out after a few seconds. The timeout is often based on the age of the file; for files that have not recently been modified, the timeout is extended.

Experience with NFS has shown that such weak consistency usually sufficient, because relatively few applications depend on strong consistency for unopened files. (Weak attribute-consistency on open files causes errors when two client hosts simultaneously attempt to append to the same file, since they have an inconsistent view of the length of the file.)

It is possible to support strong consistency in Spritely NFS, by providing a mode of the *open* RPC that says "I am reading the attributes of this file, not its data." A client would be allowed to open a file for attributes-read even if it were not allowed to open it for data-read, just as in the local Unix file system.

Any attempt to change the attributes of a file would cause the server to do an implicit "open for attributes-write," causing the appropriate callbacks to invalidate cached attributes. That is, there is no explicit open-for-attribute-write operation, because explicit attribute-writes are rare. Implicit attribute modification, such as a file length change caused by a *write* RPC, does cause a cache-invalidation callback, but the invalidation should happen only once per open-for-attributes-read.

A client will often want to read the attributes for all the files in a directory, and it does not make sense to read each of them one using a separate RPC. For this reason, and also to streamline the recovery process (see section 6.3), the Spritely NFS *open* RPC could allow "batched" operation, taking a list of files to open rather than just one. (Note that the *open* RPC returns the current attributes, so there is no need to do a subsequent *getattr* RPC.)

John Ousterhout suggests that a new form of the *readdir* RPC be added, which instead of just returning a set of file names would return a set of (*file name*, *file attributes*) pairs. An attempt by another client to modify any of the files would cause a callback referring to the directory, rather to the file itself. This might be tricky to implement because the server would have to maintain a

map between files and the directories that reference them, and the client would have to manage two different kinds of cached attributes.

A more straightforward approach would be to use a new *statdir* RPC, which returns a set of (*file name, file attributes, file handle, caching info*) tuples. Effectively, the client would open all the files in the directory for attributes-read access, and would obtain all the attribute values at the same time. Given the 8k byte limit on NFS RPC packets, in one RPC a client could read all the names and attributes for a directory containing about 60 entries. The problem with this approach is that it forces the client to close each of these file handles at some subsequent point. A batched form of the *close* RPC would avoid excess network traffic, although the client code to decide when to close a set of attributes might be rather involved.

It is not clear that strong consistency is really necessary, and it appears to be expensive to provide. However, it is feasible and if supported by the protocol, it could be enabled at the client's option. (A client that does not participate would not cause inconsistencies in other client caches, since all explicit attribute changes are write-through.)

4.4. Unmounting a server filesystem

A system manager sometimes must ''unmount'' a local file system at a file server, either to work on that file system or to cleanly shut down the entire system. With NFS, this is no problem; the unmounted file system looks like a dead server, so the client simply keeps trying. With delayed-write-after-close, however, one would like the client to be able to get the dirty blocks onto the disk before unmounting it (lest the client go down before the disk is mounted again).

More generally, the clients should discover that the file system is being unmounted, so that the server can release file references for those files that are no longer actually open. In standard Unix local-filesystem practice, one cannot unmount a file system with active references. One could preserve this behavior in Spritely NFS, but that would make system management more complex (although Spritely NFS, unlike NFS, can tell the system manager exactly which clients are using a file system). Or, one could implement an option to the unmount operation that would force clients to close their open files.

Since a Spritely NFS server can issue callbacks to its clients, it is fairly easy to implement whatever policy seems most appropriate. When the local file system receives an unmount request, it should call a special routine in the Spritely NFS server to say ''this file system is about to be unmounted.'' Spritely NFS can then do callbacks to all clients with open files on that file system, simply to force them to write back their dirty blocks, or to force them to close their files. Once this is done, Spritely NFS returns control to the local-disk file system, which can then proceed with the unmount operation.

One possible approach is that the ''unmount'' callback causes the clients to act as if the server for this file system is down. Once the disk is remounted, the server can then enter into a simplified version of the recovery protocol (described in section 6.3); in the meantime, the server can release all its state-table references into the file system (because they will be rebuilt during the recovery phase, just as if the server had crashed and rebooted).

5. Overview of the recovery protocol

Several different recovery mechanisms might have been used for Spritely NFS. The original recovery mechanism used in Sprite [Welc89] depends on a facility implemented in the RPC layer, that allows the clients and servers to keep track of the up/down state of their peers. When a client sees a server come up, the Sprite layer then reopens all of its files.

This approach provides more general recovery support than is needed for Spritely NFS, and it has several drawbacks. First, it would require changes to the RPC protocol now used with NFS,

some additional overhead on each RPC call, and some additional timer manipulation on the client. In other words, it complicates the client implementation, which is something we wish to avoid. Second, recent experience at Berkeley [Bake91c] has shown that such a "client-centric" approach can cause massive congestion of a recovering server. Sun RPC has no way to flow-control the actions of lots of independent clients (a negative-acknowledgement mechanism was added to Sprite's RPC protocol to avoid server congestion [Bake91c]).

Third, the server has no way of knowing for sure when all the clients have contacted it; even if all the clients actually respond quickly, the server still must wait for the longest reasonable client timeout interval in case some client hasn't yet tried to recover. This can make fast recovery impossible. Fourth, if a partition occurs during the recovery phase, partitioned clients may never discover that they have inconsistent consistency state.

Another possible approach is the "leases" mechanism [Gray89]. A lease is a promise from the server to the client that, for a specified period of time, the client has the right to cache a file. The client must either renew the lease or stop caching the file before the lease expires. Since the server controls the maximum duration of a lease, recovery is trivial: once rebooted, the server simply refuses to issue any new leases for a period equal to the maximum lease duration. A server will renew existing leases during this period (which works as long as clients do not cheat); the clients will continually retry lease renewals at the appropriate interval. Once the recovery period has expired, no old lease can conflict with any new lease, and so no server state need be rebuilt.

The problem with leases is that they do not easily support write-behind. Consider what can happen if a client holding dirty data is partitioned from the server during the recovery phase (not an unlikely event, since a network router or bridge might be knocked out by the same problem that causes a server crash), or if the server is simply too overloaded to renew all the leases before they expire. In such a case, the client is left holding the bag; the server will have honored its promise not to issue a conflicting lease, but will not have given the client a useful chance to write back its dirty data before a conflict might result.

Another potential problem with leases is that the duration of a lease is a parameter that must be chosen by the server. The correct choice of this parameter is a compromise between the amount of lease-renewal traffic and the period during which a recovering server cannot issue new leases, and it is unlikely that the average system manager will be able to make the right choice. The original Sprite protocol has a similar parameter, the interval between "are you alive" null RPCs, which again trades off extra traffic against the duration of the recovery phase. We would like to avoid all unnecessary parameters in the protocol, since these force people to make choices that might well be wrong. (Also, timer-based mechanisms require increased timer complexity on the client.)

The current proposal is a "server-centric" mechanism, similar to one being implemented for Sprite, that relies on a small amount of non-volatile state maintained by the server [Bake91a]. The idea is that in normal operation, the server keeps track of which clients are using Spritely NFS; during recovery, the server then contacts these clients and tells them what to do. Since the recovery phase is entirely controlled by the server, there is less chance for congestion (the server controls the rate at which its resources are used). More important, the client complexity is minimal: rather than managing timers and making decisions, all client behavior during recovery is in response to server instructions. That is, the clients require no autonomous "intelligence" to participate in the recovery protocol.

For this to work, the use of stable storage for server state must be quite limited, both in space and in update rate. The rate of reads need not be limited, since a volatile cache can satisfy those with low overhead. Stable storage might be kept in a non-volatile RAM (NVRAM) (such as

PrestoServe™), but if the update rate is low enough it is just as easy to keep this in a small disk file, managed by a daemon process. Updates to this disk file might delay certain RPC responses by a few tens of milliseconds, but (as you will see) such updates are extremely rare.

6. Details of the recovery protocol

6.1. Normal operation

The stable storage used in this protocol is simply a list of client hosts, with a few extra bits of information associated with each client. One is a flag saying if this client is an NFS client or a Spritely NFS client. Only Spritely NFS clients participate in the recovery protocol, but we keep a list of NFS clients because this could be quite useful to a system manager. Another flag records whether the client was unresponsive during a recovery phase or callback RPC; this allows us to report to the client all network partitions, once they are healed.

During normal operation, the server maintains the client list by monitoring all RPC operations. If a previously unknown client makes an *open* RPC, then it is obviously a Spritely NFS system. If a previously unknown client makes any file-manipulating RPC before doing an open, then it is an NFS client. One can devise a set of procedures to handle the cases of clients that start out using one protocol and switch to the other (perhaps as the result of a reboot); that is covered in section 7.1.

The client list changes only when a new client arrives (or changes between NFS and Spritely NFS). This is an extremely rare event (most servers are never exposed to more than a few hundred clients) and so it does not matter how expensive it is. My assumption is that it is comparable in cost to a few disk accesses; that is, not noticeable compared to the basic cost of an *open* or file access.

On the other hand, the server must check the cached copy of the client list on almost every RPC. This should be doable in a very few instructions, if the client list is kept in the right sort of data structure (such as a broad hash table). The overhead should be less than is required to maintain the usual NFS transaction cache.

Note that the server's volatile copy of the client list need not contain the entire list of clients, but could be managed as an LRU cache, as long as it is big enough to contain the working set of active clients. This might conserve memory if there are a lot of inactive clients on the list.

If a client fails to respond to a callback (or during the recovery phase, described later) then the server marks it as "embargoed." This could be because the client has crashed, but it might be because the client has been partitioned from the server. When an embargoed client tries to contact the server, the server responds with a callback RPC to inform the client that it was partitioned during an operation that might have invalidated its consistency state; once the client replys to this callback RPC, the server clears the embargoed bit[5]. The client thus knows that its state is inconsistent, and can take action to repair things (or at least report the problem to the user).

Some additional design work is required to figure out how best to reestablish a consistent state after an embargo is lifted. One approach would be to follow a "scorched earth" policy, in which both the client and server return to an initial condition. However, it is probably possible (using generation numbers) for a client to detect which of its open or cached files are still consistent, and only scorch those files which actually have conflicts. The "reintegration" techniques used in the Coda system [Saty90] might also prove useful.

---

[5]And returns an error response to the original RPC?

6.2. Client crash recovery

When a client crashes and reboots, the server will be left believing that it has files open even though it does not. This could lead to false conflicts and thus reduced caching. The server will discover some false conflicts when it does a callback and the client says "but I don't have that file open." In other cases the false conflict would not cause a callback (i.e., if caching is already disabled). Also, these "false opens" clutter up the server's state table and space reservation count.

A client can ameliorate these problems by issuing a special "I just rebooted" RPC the first time it mounts each file system after a reboot. The server uses this RPC to force a close on all the files that were open from that client, and also to reclaim all disk space reserved for that client's dirty blocks. (If the client uses non-volatile RAM technology to keep dirty blocks across a crash, it should also use it to preserve a list of all dirty file handles, and write back the dirty blocks before sending the "I just rebooted" RPC.)

If a client reboots while a server is down (or unreachable because of a network partition), the client simply keeps retrying its mount operation until the server recovers (or becomes reachable), just as is done in NFS.

6.3. The server recovery phase

When a server crashes and reboots, it enters a recovery phase consisting of several steps. The server herds the clients through these steps by issuing a series of recovery-specific callback RPCs, each of which requires a simple response from the client. The steps are:

1. **Read the client list from stable storage**: The server obtains the client list from stable storage and loads it into a volatile cache.
2. **Initiate recovery**: The server contacts each Spritely NFS client on the client list[6]. The callback tells the client that the recovery phase is starting; until the recovery termination step is complete, clients are not allowed to do new opens or closes, and cannot perform any data operations on existing files.

   When a client responds to this RPC, the server knows that the client is participating in the recovery protocol; clients that do not respond are marked as embargoed. During the rest of recovery, embargoed clients are ignored and we can assume that the other clients will respond promptly, but during this step long timeouts may be needed. On the other hand, this step can be done for all clients in parallel, so the worst-case length of this step is only slightly longer than the maximum timeout period for deciding that a client is down or partitioned.

   At the end of this step, we can update the stable-storage client list to reflect our current notion of each client's status.
3. **Rebuild consistency state**: The server contacts each non-embargoed client and instructs it to reopen all of the files that it currently has open, using the same open-mode (read-only or read-write) that was used before. If the clients do not cheat, the resulting opens will have no conflicts, since before the server crashed there were no conflicts and no new opens could have taken place since the crash.

---

[6]Should it contact embargoed clients? If so, then we need to let the client know at this point that it has already been embargoed, and that will complicate the rest of the recovery process. If not, then the damage from a partition may be compounded even further. Simplicity, and an aversion to incurring timeouts, suggests that we should not bother to contact already-embargoed clients at this point.

Since each server may have to open multiple files, and since file-open operations are moderately expensive (requiring manipulation of the state table), the server may want to do these callbacks serially rather that in parallel (or semi-parallel, to limit the load to a reasonable value). This should not result in too much delay, since we are reasonably sure that the clients involved will respond.

Note that a client may hold dirty blocks for files that it does not actually have "open". This means that there must be a special mode of the *open* RPC, used only for "reopening" closed-dirty files. It might be better to define a special *reopen* RPC that allows the client to reopen several files in a single network interaction. Otherwise, the RPC layer could become the bottleneck during recovery. (Or, the basic *open* RPC could allow batched operation, which might be useful in providing consistent attributes caching as in section 4.3.)

A client responds to this callback only when it has successfully reopened all of its open files. If a client fails to respond, then the server marks it as embargoed and updates the stable-storage list.

Once this phase is done, the server has a complete and consistent state table, listing all of the open and closed-dirty files.

4. **Terminate recovery**: The final step is to contact each client to inform it that recovery is over. Once a client receives this RPC, it can do any operation it wants. As in the recovery initiation step, the server can do these callbacks in parallel, but in any case the clients are unlikely to timeout so the duration of this step should be brief.

One issue that I have not yet considered is what might happen if a server crashes during its recovery phase. It may be that a simple crash-generation number scheme, passed with the crash-recovery callbacks, will allow the clients to keep track of what is really going on.

6.4. Log-based recovery

Because the recovery protocol is server-centric, it leaves the implementor of the server a lot of freedom to choose different strategies. V. Srinivasan has pointed out that nothing in the protocol prevents the server from using additional stable storage to obviate part or all of the recovery protocol.

The server could, for example, log all opens and closes to stable storage. Since the "open lifetime" of files is fairly short (often less than 100 milliseconds [Bake91b]) it would not make sense to log every such event to disk. Instead, the server could keep the head of the log in NVRAM, which would allow it to elide the short-lived opens before writing the log to the disk. Some sort of log-cleaning algorithm, analogous to that required by a log-structured file system [Rose91], would be necessary. Alternatively, the on-disk information could be structured as a database, which would take more work to update but which would not need cleaning. Using the log or database, the server could recover its consistency state without any help from the clients.

A much simpler approach would be to keep track, in the client list, of those clients that have any files open at all. During crash recovery, the server could ignore any client known to have no open files, thus speeding recovery and perhaps avoiding timeouts for clients that have been removed from service. This modification would increase the update rate for the stable-storage copy of client list. However, the server could delay the update on a client's last close, anticipating a subsequent open in the near future, because this would not affect the correct behavior of the recovery protocol. A delay interval of, say, one minute would probably avoid almost all extra updates without significantly increasing the cost of recovery.

6.5. Recovery and the ENOSPC problem

Since the server host's count of reserved disk blocks may be updated quite often, it does not make sense to keep it in stable storage. (Maintaining a stable accurate value could approximately double the latency of disk writes for closed-dirty files.) Instead, we can recompute this value during the recovery phase. When recovery starts, we set the value to zero. We then have the clients tell us what their reservation needs are, either by an extra argument to the RPC call for "reopening" closed-dirty files, or perhaps by having the client also explicitly close those files (since the *close* RPC always carries a reservation request). Once recovery is done, we have a consistent count of the total reservation requirements.

Note that during recovery, a client cannot simply request a reservation for the number of dirty blocks it currently holds, because this number might have increased since the server crashed. Instead, the client must remember the reservation it has left as the result of a normal *close* RPC, and use this value when "re-opening" a closed-dirty file.

If the network is partitioned during recovery, we might end up in a state where the server does not know of a client's reservation requirements, and so gives the space away once recovery is over. If the partition heals, we may discover that no conflicting open prevents the embargoed client from writing its dirty blocks, but there is no longer any space to hold them.

One (rather crude) approach to this problem is to set aside some disk space in anticipation of this problem. For example, some file systems, such as the Berkeley Fast File System [McKu84], reserve a certain amount of free space in order to obtain better performance. This so-called "minfree zone," which can be used by super-user processes on normal Unix systems, might also be employed to store blocks written back from embargoed clients. However, this is at best a stop-gap solution and can lead to some tricky management problems: what do you do when this space runs out?

6.6. Multiple file systems exported by one server host

NFS servers traditionally export more than one file system; a large server might export dozens of file systems. This allows a network manager to reconfigure servers without broad disruption, and to tailor security controls according to the nature of the data being protected.

The recovery protocol described in this paper is most simply understood as applying to each individual file system. That has two implications:

1. One client list is maintained for each file system, not just one for each server host.
2. The steps in the recovery protocol must be repeated for each file system exported by a server. (The server should not attempt to contact clients that failed to respond during an earlier iteration.)

The latter requirement is not actually a serious problem. The initial and final phases, during which all clients are contacted in parallel, can be done in parallel for the various file systems. The number of packets exchanged in these steps will increase, but the number of packets exchanged as the clients reopen their files will not be affected.

The former requirement increases the amount of storage space needed to keep client lists, both stable storage and in-memory cache. Since these lists are not likely to be very large, keeping one for each file system should not be too wasteful. The alternative, keeping one list for the entire server, could add run-time complexity, since it is conceivable that a given client might mount one file system via Spritely NFS, and another file system from the same server via pure NFS. This would force the server to use a complex data structure to represent a client list, reducing the incentive to keep one list instead of several.

It is tempting to try to avoid these requirements by making the recovery protocol messages express things in terms of server hosts rather than server file systems. That is, the three callback

types used in the recovery process inform a client that it should perform a particular function for all the files opened from a specified server host, rather than a specified file system. I believe this will work, but some problems might be lurking in the background.

7. Simultaneous mixing of NFS and Spritely NFS hosts

I argued that without a path for incremental adoption, users will have little incentive to install Spritely NFS, because all-at-once changeovers cause major disruption. A sudden change to a new, untried system makes system managers nervous.

Spritely NFS clients and server can easily coexist with pure NFS hosts. The two problems to solve are automatic configuration (so that network managers need not worry about who is running what) and maintenance of consistency guarantees (so that NFS clients get at least the level of consistency that they would if all clients were using NFS).

7.1. Automatic recognition of Spritely NFS hosts

It is possible for Spritely NFS clients and servers to "recognize" each other in a sea of NFS hosts. Suppose that Spritely NFS were to use the same RPC program number as NFS; we can establish a set of rules that will allow Spritely NFS hosts to discover if their peers speak Spritely NFS or just NFS.

Consider a Spritely NFS client. It need not know if the server supports Spritely NFS (i.e., cache-consistency protocols) until it wants to open a file. At that point, it simply issues its *open* RPC. If the server speaks only NFS, it will respond to this with a PROC_UNAVAIL error code. The client can then cache this fact (in a per-filesystem data structure) and treat the file system as a pure NFS service.

A Spritely NFS server recognizes Spritely NFS clients because they issue *open* RPCs before issuing any file-manipulating RPCs. Thus, when a new client is added to the server's client list, the RPC that causes this addition also tells the server what kind of client is involved.

If we want to follow this "automatic" scheme for recognizing Spritely NFS hosts, then it should work even with a client or server changes flavors. In principle, client changes should be easy to detect, since a client changing from NFS to Spritely NFS will issue an *open* RPC, and the server can check on each *open* to make sure that its client list records the client as a Spritely NFS host. A client changing back to NFS would reveal itself, sooner or later, by using a file that it had not previously opened. This trick requires the server to check a file's consistency state on every RPC, which is otherwise unnecessary (this is normally the job of the client) and could add some slight overhead.

If a Spritely NFS server changes back to an NFS server, the Spritely NFS clients will detect this as soon as they do an *open* or *close* operation. If an NFS server changes into a Spritely NFS server, however, the clients might not realize this immediately. It might be possible for the server to signal its nature by the use of a callback, but this could cause problems to pure-NFS clients that are not expecting any callbacks.

Another approach, instead of basing automatic recognition on RPC procedure types, is to use a separate RPC program number for Spritely NFS. This makes the server's task a lot easier; it simply distinguishes clients based on which program number they use. The server would not have to check to see if a client had previously opened a file in order to catch transitions between Spritely NFS and pure NFS. This does not, however, solve the problem of how a client realizes that a server has changed from NFS to Spritely NFS.

7.2. Consistency between NFS and Spritely NFS clients

When NFS and Spritely NFS clients are sharing a Spritely NFS file system, the NFS clients will not have the same consistency guarantees as Spritely NFS clients. However, the Spritely NFS server can guarantee the NFS clients no-worse-than-NFS consistency, by treating each NFS operation as if it were bracketed by an implicit pair of *open* and *close* operations.

In other words, if an NFS client reads from a file which is write-cached by a Spritely NFS client, the server first does a callback on the Spritely NFS client to obtain the dirty blocks. If an NFS client writes a file cached by a Spritely NFS client, the server does a callback on the Spritely NFS client to invalidate its cache. If this reduces performance too badly, perhaps Spritely NFS callbacks should optionally specify a particular block to flush or invalidate ... but I suspect that if such NFS-to-Spritely-NFS write-sharing happens at all, then it is likely to involve most of the blocks in a file.

8. Performance

Our original goal with Spritely NFS was to improve performance over NFS. Since NFS does not need to support a recovery protocol, we must show that the added recovery overhead in Spritely NFS does not eliminate our advantage. Note that the original, non-recovering version of Spritely NFS did better than NFS on realistic benchmarks even though NFS does not have to do any *open* and *close* RPCs; that is, Spritely NFS saves enough through better use of the client cache to make up for the extra RPCs.

The recovery protocol has two kinds of costs: in normal operation, there is a small overhead on each RPC, and after a server crash, there is a recovery phase. Since NFS has no recovery phase, it will always be faster at continuing after a server reboot. These should be rare events, so the cost of recovery will be amortized over a long period of useful work. At any rate, the server-centric approach should allow us to do efficient recovery, since we are not put at risk of server overload during the recovery phase.

The per-RPC overhead comes from the maintenance of the client list. I argued earlier that this is negligible; most of the time, we simply do a hash-table lookup to discover that the client is already known and not embargoed[7]. Very rarely, we must update stable storage, but it is unlikely that a server would see such a high rate of new clients that this becomes a measurable overhead. In short, I do not think the per-RPC overhead will cause a measurable difference in Spritely NFS performance.

9. Software complexity

Since I have described this recovery protocol as "simple," it seems appropriate to describe how much work it would take to convert an NFS implementation to support Spritely NFS with recovery. Note that the original Spritely NFS implementation was written in the course of a month or so by a programmer who had never before studied the Unix kernel. See [Srin89] for details.

---

[7]The per-RPC operations in the original Sprite recovery mechanism apparently made a small but measurable difference in the RPC overhead. This might have been because on each RPC request and reply, the code was forced to manipulate timers.

9.1. Client implementation issues

Starting with a client NFS implementation, the modifications necessary to support Spritely NFS are fairly simple. The *open* and *close* operations have to be implemented, the per-file data structures need to include cachability information, and the data access paths need to observe the cachability information. A daemon, patterned closely on the existing NFS server daemon, needs to be added to handle callback requests, along with "server" code to respond to the callbacks.

Very few changes are needed in other components of the client operating system. The code that manages the table of open and closed files (the equivalent of the Unix *inode* table) must inform the Spritely NFS client when a closed file is being removed from this table to make room for a new entry. It is also useful to provide a mechanism to remove dirty blocks from the file cache, for use when the file that contains those blocks is deleted (this improves performances by eliminating useless write-backs).

9.2. Server implementation issues

The changes to the server are obviously more extensive. For Spritely NFS without recovery, the changes were quite localized: the existing RPC server procedures were not touched, and all the new code related to handling the *open* and *close* RPCs and performing callbacks. Spritely NFS requires a small amount of stable storage to support the "generation number" mechanism used to detect certain conflicts. One 64-bit value kept per file system (or even per server), and updated every hour or so, should suffice.

To support recovery, the server code for all NFS operations has to check the client list on each RPC, and perhaps call functions to maintain the client list or do necessary callbacks.

Most of the complexity in the recovery protocol can be implemented in user-mode code. The Spritely NFS kernel code would have to provide some hooks so that the recovery process can disable the servicing of certain RPCs during the recovery phase.

In order to provide full consistency between Spritely NFS clients and local file system applications running on the server, there will have to be some linkages between the local file system's *open* and *close* operations and the Spritely NFS state-table mechanism. For example, when a local process opens a file, this might require Spritely NFS to change a client's cachability information for that file. The local file system must also support the disk-space reservation scheme described in section 4.1; this means providing a special form of the *write* operation that decrements the reservation.

10. Other related work

Several interesting papers related to recovery in distributed file systems have never been published. Rick Macklem worked on "Not Quite NFS," an attempt to use the leases model to provide recovery for an NFS extended with a Sprite-like consistency protocol. Meanwhile, the Echo file system project at Digital's Systems Research Center has grappled with a number of similar issues, especially those related to write-behind [Birr92, Hisg89, Swar92].

Mary Baker and Mark Sullivan describe a similar approach to state recovery [Bake92], using a "recovery box": stable storage for selected pieces of system state, to allow a system to reboot quickly. In their approach, a file server would store all the open file handles in stable storage, with the assumption that these are unlikely to be corrupted by (or just prior to) a crash. My proposal is more conservative, both in that it does not require low-latency stable storage, and because it makes far weaker assumptions about the effects of a crash. Their proposal, however, leads to much quicker recovery.

11. Summary

Spritely NFS was an interesting experiment, but without a recovery protocol it is not suitable for production use. The recovery protocol proposed in this paper, together with tying up some loose ends, should be enough to make Spritely NFS a real alternative to NFS. The mechanism is so simple, especially on the client side, that one can no longer claim that only a stateless protocol admits a simple implementation.

Even if Spritely NFS never becomes a real system, I believe that this bare-bones approach to recovery will be useful in other contexts. A similar approach is being used now in Sprite, and their experiences should validate the design.

Acknowledgements

The design in this paper has evolved (sometimes rather discontinuously) in lengthy exchanges among many people, including (in alphabetical order) Mary Baker, Cary Gray, Chet Juszczak, Rick Macklem, Larry McVoy, John Ousterhout, V. Srinivasan, Garret Swart, and Brent Welch. Most of these people have talked me out of at least one bad idea.

References

[Bake91a] Mary G. Baker. Private communication. 1991.

[Bake91b] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 198-212. Pacific Grove, CA, October, 1991.

[Bake92] Mary Baker and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment. In *Proc. Summer 1992 USENIX Conference*. San Antonio, Texas, June, 1992. To appear.

[Bake91c] Mary Baker and John Ousterhout. Availability in the Sprite Distributed File System. *Operating Systems Review* 25(2):95-98, April, 1991.

[Birr92] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo Distributed File System. In preparation. 1992.

[Gray89] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. 12th Symposium on Operating Systems Principles*, pages 202-210. Litchfield Park, AZ, December, 1989.

[Hisg89] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 49-53. Pacific Grove, CA, September, 1989.

[Jusz89] Chet Juszczak. Improving the Performance and Correctness of an NFS Server. In *Proc. Winter 1989 USENIX Conference*, pages 53-63. San Diego, February, 1989.

[McKu84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems* 2(3):181-197, August, 1984.

[Nels88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.

[Oust85] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 15-24. Orcas Island, WA, December, 1985.

[Rose91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation ofa Log-Structured File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 1-15. Pacific Grove, CA, October, 1991.

[Saty90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers* 39(4):447-459, April, 1990.

[Shir92] Ken W. Shirriff and John K. Ousterhout. A Trace-Driven Analysis of Name and Attribute Caching in a Distribute System. In *Proc. Winter 1992 USENIX Conference*, pages 315-331. San Francisco, CA, January, 1992.

[Srin89] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proc. 12th Symposium on Operating Systems Principles*, pages 45-57. Litchfield Park, AZ, December, 1989.

[Swar92] Garret Swart, Andrew D. Birrell, Andy Hisgen, and Timothy Mann. The Failure Semantics of Write Behind. In preparation. 1992.

[Welc89] Brent B. Welch. *The Sprite Distributed File System.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California—Berkeley, 1989.

An Efficient Variable-Consistency Replicated File Service

Carl D. Tait and Dan Duchamp

Computer Science Department
Columbia University
New York, NY 10027
{tait,djd}@cs.columbia.edu

ABSTRACT

In a previous paper, we introduced a new method of file replica management designed to address the problems faced by mobile clients. Two goals shaped our design: minimizing synchronous operations, and letting clients determine the level of consistency desired. The previous paper was a high-level view of the most fundamental issues; here, we refine our previous ideas and flesh out the rest of the design.

1 Introduction

This work investigates how to design a replicated file system that will serve mobile clients well. The idea that file service clients might be mobile is a natural one, given the likely marriage between two exploding trends: portable computers and wireless networks.

In a previous paper [14] we have argued that client mobility is a major new development that requires re-thinking file system design, and that existing approaches to replica management (e.g., [7, 13]) would not cope well with mobile clients. In that paper we proposed an alternative: a lazy "server-based" update scheme and a new service interface that allows applications to select strong or weak consistency semantics on each particular read call. Our earlier work addressed only some of the aspects we see in the complete problem, and developed our solution in only a certain amount of depth. This paper finishes our design.

We make several assumptions about operating conditions:

1. Client movements cannot be constrained, although patterns of movement may exist.
2. Latency of remote operations degrades as the distance between hosts increases.
3. The presented load is what Ousterhout has called "engineering/office applications" [10]. In this model workload, sequential sharing is not uncommon, but simultaneous sharing (other than read-read) is rare.
4. A file cache of modest size is maintained by each client.

5. File service sites can synchronize their clocks. This assumption, once controversial, can now be satisfied by several clock synchronization protocols (e.g., [8]). We do not make any assumptions about client clocks.

Flowing from these assumptions is our conclusion that three design goals are of paramount importance in handling mobile clients: we must minimize synchronous multi-server operations, ease the addition and deletion of server sites, and allow for incomplete replication at servers. We discuss our reasoning for each in turn.

Minimize synchronous multi-server operations. Our first two assumptions lead us to the conclusion that file systems that frequently use "global communication"[1] will not perform well if clients move over a wide area while they remain mapped to a fixed set of servers. The reason is that the latency of contacting the most distant server is a lower bound on the latency of the entire multi-server operation. File service operations that involve global communication will slow down when a client moves away from its current set of servers.

Ease the addition and deletion of server sites. In order to avoid global communication, two features seem desirable. First, a primary-secondary server organization, in which the client communicates only with the primary and the primary communicates asynchronously with the other servers. Second, ensuring that the primary is always located near the client. Since the client moves unpredictably, the service must be ready to regularly add a new replication site, *typically as the new primary.* The model of mobile operation we envision is that a client moves over a wide area, perhaps to areas it has never visited before. Once in a new milieu, it will negotiate with some local machine to become its new primary. There is also presumably a need for regular replica deletion, in order to limit the degree of replication to a sensible number in case of a highly mobile client. Regular addition and deletion of replicas, especially the primary, marks our design as unusual.

Allow for incomplete replication. If identities of the servers — specifically, the primary — are changing frequently and unpredictably, replication sites should not be expected to always store complete copies of the client's file set. We provide for incomplete replication, in which a client's files need be replicated at only a subset of the server sites.

Our previous paper addressed only the first of these concerns, and did so in limited detail. In this paper, we address the goals of replica site addition/deletion and incomplete replication, while refining and expanding our previous ideas. In particular, we describe how our design manages the changing relationships between clients, servers, and files, and provide a careful analysis of failure cases.

2 Operation in the Absence of Failures

As suggested in the introduction, our design reduces global communication by using a primary-secondary server hierarchy. The client communicates synchronously with the primary only. We employ write-back caching with the primary, not the client, choosing when updates are copied from the client's cache. The primary makes periodic *pickups* from the clients it is servicing, and propagates updates back to the secondaries asynchronously. This allows the client's `write()` operation to return immediately after placing the new value

[1] We define a global communication as an operation in which several servers must be contacted synchronously. Coda's `close()` operation is an example in this category.

in its cache.

Once some number, N, of secondaries have acknowledged receipt of an update, the primary informs the client that the associated cached update can be discarded. This notification — which might be piggybacked onto the next pickup request — is called a *purge notice.* Because a client must retain an update in its cache until receiving a purge notice, the service has some latitude (constrained by the size of the client's cache) to "wait out" secondary server failures without any disruption in accepting `write()` calls.

During periods of heavy update activity, a client's cache may fill between primary server pickups. For this case, we provide a synchronous, blocking *forced write* operation that causes an immediate pickup.

Asynchronous update propagation has well-known benefits and dangers, and it has been used in the design of other replicated file systems (e.g., [13]). Our work is significantly different in two ways. First, by requiring that the client retain an update until a sufficient number of secondaries have the new value, we trade cache space for a low-latency, yet reliable write operation. Other systems typically block while some number of replicas are written (the remainder being written asynchronously), or return immediately with no guarantee that the update will be safely propagated. We push back the blocking point so that the client typically is not involved in the propagation process.

The second feature that sets our work apart is how we couple asynchronous update propagation with a read interface that allows an application to choose either "lazy" or "UNIX-like" semantics on a per-read basis, as described below. When there is little sequential sharing, this design achieves high fault tolerance and low latency for both reads and writes. Experimental data presented in our earlier paper supports the feasibility and potential value of this dual-read-call interface.

Given such a design, the key issues are:

- How does this interface work?
- When, why, and how do clients and primaries bind to each other?
- How are files replicated on secondaries?

We address these issues in the next three subsections.

2.1 Dual-Read-Call Interface

We split the traditional `read()` interface into `strict_read()` and `loose_read()`. There is no guarantee concerning the value returned by the loose form, whose implementation is shown in Figure 1. In principle, the strict form returns the "most recent consistent" value; this will be defined more precisely after we have explained the operation of `strict_read()`. If `strict_read()` and `write()` are used exclusively, the system provides a replicated analogue of "one-copy UNIX semantics" (1USR): the semantics found in a centralized UNIX system. 1USR is not equivalent to one-copy serializability (1SR): due to caching, 1USR clients can make conflicting updates to a file without knowing that they are doing so. For compatibility, a generic `read()` library call could simply call `strict_read()`. For the convenience of some applications, another library routine could be defined with

```
Look in the client's cache

If there is no copy in the cache, then check the primary server

If there is no copy at the primary, then check any of the secondaries,
   in any order
```

Figure 1: Loose Read Algorithm

the following semantics: strictly read the requested file if possible; otherwise, return any available copy.

While `loose_read()` is allowed to return any convenient value, analysis of file-trace data presented in our previous paper indicates that a "best effort" implementation will almost always return the most up-to-date value. This conclusion is supported by the recent study by Baker et al. [1], which found that only one third of one percent of `open()` calls read data that was written by another client less than 60 seconds previously. This study concluded that (automatic) cache consistency is desirable to prevent clients from using stale data; in our system, clients use the `strict_read()` operation to enforce cache consistency.

2.1.1 Currency Tokens

A naive implementation of `strict_read()` would contact all servers and all clients that had read the file and retrieve the most up-to-date copy it finds. *Currency tokens* (CTs) are used to avoid this naive approach.

CTs rely on the idea of a *potential consistent writer*, or PCW. A PCW is a client site with a process that has strictly read a file, and that has write permission for that file. In other words, a client is declared a PCW after demonstrating both desire (strict read) and ability (write permission) to make an update in a consistent fashion. A CT is given in response to a `strict_read()` if there are no PCWs for that file, or if the client is the only PCW. A client can receive a CT only from a strict read, never from a loose read. Holding a CT allows the client to know that either there are no PCWs for the file in question, or that all potential updates are localized to itself and its primary-secondary hierarchy.

A client that performs a strict read without a CT initiates a relatively complex series of actions. First, recall that an update must be replicated on at least N secondaries before a client is allowed to purge the update from its cache. Therefore, assuming that there are a total of T secondaries, at least T-N+1 secondaries must be contacted — as well as any PCWs that have the file in their caches — to ensure that the most recent value is located. This condition is simply that of quorum consensus, wherein the read and write quorums must overlap [4].

Files are tagged with timestamps so that the copies at different replicas can be compared for recency. After reading T-N+1 copies, the primary compares the timestamps and gives the client the most recent copy of the file plus an indication of whether it has a CT. To ensure consistency, lists of PCWs and clients who hold CTs must be maintained in non-

```
Client sends strict_read() to primary

Phase 1:
   Primary multicasts strict_read() to all secondaries

   Each secondary evaluates whether client is a PCW, and,
      if so, records the fact in non-volatile storage

   Each secondary returns a timestamped file to the primary, along with
      timestamped lists of PCWs and CTs for this file (if there are any)

Phase 2:
   Block until at least max{T-N+1, majority} secondaries reply

   Using the most up-to-date PCW list it has received,
      the primary determines that client should be given a CT
      iff there are no PCWs, or client is the only PCW

   If a new CT is being granted, this fact is recorded in
      non-volatile storage on at least a majority of secondaries

   If old CTs must be revoked due to the arrival of a PCW,
      the primary performs the revocation

Phase 3:
   Primary reads cached copies from all PCWs

   Primary returns most up-to-date replica gathered from
      secondaries and PCWs, along with a CT if appropriate
```

Figure 2: Strict Read Without a CT

volatile storage on at least a majority of secondaries so that this information will be found during any initial `strict_read()`. The complete algorithm is sketched in Figure 2.

Subsequent strict reads with a currency token are considerably simpler and faster. The algorithm for this case is shown in Figure 3; note the similarity to the loose read algorithm in Figure 1. CTs have the effect of allowing most strict reads to be implemented by executing almost the same sequence of actions that is performed for a loose read. The only difference is at the level of secondaries: if the search gets that far, a strict reader with a CT must contact enough secondaries to ensure that the most recent copy is located. The initial strict read is the only point in our design at which a synchronous operation is required: several secondaries are contacted, as well as all PCWs.

PCW and CT lists are updated at secondaries by a simple process that can do only one update at a time. It is necessary to wait in line to have a PCW or CT added to (or

```
Look in the client's cache

If there is no copy in the cache, then check the primary server

If there is no copy at the primary, then check at least T-N+1
   secondaries and return the most recent value found
```

Figure 3: Strict Read With a CT

OPERATION	QUORUM REQUIREMENT
Initial strict read	read file at T-N+1 secondaries read/write PCW and CT lists at majority of secondaries
Strict read with CT	read file at T-N+1 secondaries if secondaries read at all

Table 1: Quorum Conditions for Strict Read

deleted from) the list. Furthermore, since we allow initial strict reads from different clients to execute simultaneously, it is necessary that read quorums intersect so that at least one client will learn about the other. Thus, initial strict reads must actually contact T-N+1 secondaries or a *majority* of secondaries, whichever is larger. A complete statement of the quorum conditions is given in Table 1.

Correctness of Simultaneous Operations. These mechanisms, combined with the fixed sequence of operations involved in gaining a CT, implicitly serialize initial strict reads. Overlapping strict reads of the same file are therefore assured of functional correctness. To see why this is true, assume that two clients start an initial strict read at the same time:

1. Two non-PCWs: No problem, of course.

2. Two PCWs: Because read quorums overlap, at least one client is guaranteed to learn about the other's existence. Whichever client finds out revokes all CTs.

3. A non-PCW and a PCW: The same reasoning as (2), with one additional tricky case. The non-PCW may update the CT list *after* the PCW has read it. Thus, the PCW must ensure that the CT list has not changed since it was read (using timestamp comparison). If the list has been modified, it is re-read and re-analyzed. This is repeated until the list stabilizes.

   (Proof that the CT list will stabilize quickly: (1) If no new strict readers arrive, the list stabilizes immediately. (2) If a new non-PCW arrives, it notices the PCW in the PCW list. This prevents a CT from being granted to the new non-PCW, so the CT list is not modified. (3) If another PCW shows up, all CTs are revoked and the list stabilizes to an empty list.)

A small complication arises because we allow *incomplete replication*; that is, there is no requirement that a file be replicated at every secondary. It is possible that some

secondaries will not have the file at all, much less the most recent version. Nonetheless, by using the scheme described above, we ensure that an initial strict read will find the most recent replica, assuming that a sufficient number of secondaries are available. A secondary that does not have a replica of the file in question acts as a kind of *witness* [11]: a server that may be counted as part of a quorum even though it contains no explicit information about a file.

Reducing Initial Strict Reads. In order to reduce the number of initial strict reads, which require global communication, CTs should apply to groups of files rather than individual files. We expand the scope of a CT by letting it cover a file's entire directory if there are no other PCWs for any files in that directory. (We are assuming the existence of a standard hierarchical name space.) In addition, a file that has several names — through symbolic links, for example — is covered by a single CT; it is not necessary to gain a CT for each name.

We have rejected the seemingly attractive notion of expanding a CT beyond the confines of a single directory. There are two major problems with doing so. First, expansion can be very time-consuming: in the general case, the number of files covered increases exponentially as we move up or down in the name space. Second, after spending time in expansion, a single additional PCW can topple the carefully constructed CT edifice. On the other hand, single-directory expansion has several advantages:

- Expanding CTs is fast and easy.
- We don't lose much if another PCW shows up.
- Directories generally contain enough files to make this approach dramatically superior to single-file CTs.

Currency Token Revocation. CTs can be revoked for several reasons. The guiding principle is that the presence of even one PCW for a file requires all other clients to give up their CTs on that file. However, we provide a "short-circuit" optimization for the single-writer, multi-reader case: reads are directed to the unique PCW. If there are two or more PCWs, no clients can hold CTs, and the short-circuit scheme is ruled out.

What happens if the service needs to contact a currently unreachable client, either to revoke a CT or to read a file from a PCW's cache? A straightforward solution involves timeouts: a client who cannot be contacted for a certain length of time is deemed to have become permanently disconnected. Record of the client's CTs and PCW status is discarded, and other users' strict reads are allowed to proceed. But this approach cannot completely close the window of error: a client who has been disconnected might suddenly reappear with a cache full of updates that were missed by strict readers in the interim.

Although we cannot eliminate errors entirely, we can at least eliminate the inelegance and delay of timeouts; we have concluded that demand-based disconnection is a convenient simplification. Whenever the service needs to contact a client, but the client is unreachable for some reason, we give up immediately, automatically revoking any CTs that the wayward client is holding. This applies with equal force whether the system is making server-based pickups, attempting to contact PCWs, or explicitly revoking CTs.

It is important to realize that CTs are not equivalent to read and write tokens used by other systems. They are simply *hints* used to improve the performance of strict reads. In other systems, tokens are prerequisites for performing operations; in our design, strict reads can take place without using currency tokens, though performance will be less than optimal. CTs are somewhat akin to *callbacks* in Coda [12], though the pessimistic nature of CTs provides a stronger consistency guarantee: a CT will not be granted if any other client has even the *potential* to make a consistent update.

2.2 Client-Primary Attachment

Client-to-primary assignments are made by a special module called the *matchmaker*, which is most conveniently implemented as a process on any service machine. When a client wishes to obtain a primary server, it sends a request to any convenient matchmaker, which then selects a primary based on criteria of its choice. For coping with mobile clients, the criterion might be physical proximity.

After making its choice, the matchmaker sends the client's address to the selected primary and forgets about the transaction. The new primary server then performs a pickup so that the client will learn the identity of its new primary. The client saves the primary's address in (volatile) storage for use in subsequent read operations.

A newly-selected primary always starts with an empty cache for the client it is servicing. This is necessary to prevent consistency problems that might occur due to currency token expansion. For example, if a client strictly reads `/usr/local/bin/emacs`, the client will typically be given a CT that covers all of `/usr/local/bin`. The primary, however, is only guaranteed to have the most recent version of `emacs`, not every file in the directory. Therefore, we must ensure that a down-level version of (for example) `gprof` is not returned during a strict read just because the primary happens to have a copy lying around.

The only non-volatile state retained by the matchmaker is a list of servers that can act as primaries. It is not necessary to modify this list when a potential primary crashes. The matchmaker can "ping" the machine it has chosen for a given client, and if there is no response, the matchmaker will simply choose a different server. Modifying the list of primaries is trivial. Adding a new server poses no problems, and deletion is not difficult. If a client attempts to use a primary that has been deleted from the list, either there will be no response, or the message will be rejected by the disgruntled ex-primary. In either case, the client need only apply to the matchmaker to obtain a new primary.

Given the relative infrequency of calls to the matchmaker, we do not anticipate any scalability problems. For availability, however, it is desirable to run the matchmaker on several different machines. Since the matchmaker is stateless except for the list of potential primaries, multiple executions on different machines can operate independently if necessary.

A client may ask the matchmaker for a new primary at any time, for any reason. Because the client does not discard cached updates until receiving confirmation that they have been stored on the required number of secondaries, switching among primaries in any arbitrary way is no threat to correctness. Similarly, a primary may safely pull out of a client-primary relationship at any time. In both cases, the client will have to obtain a new primary and pickups will resume normally.

Good reasons for requesting a new primary include failure of the old primary and

movement (leading to reduced performance) by the client. Letting the client drive the selection process substantially simplifies our design, especially in the face of failures (see Section 3.2 for details). There is a cost for changing primaries: the matchmaking protocol must be run, CTs must be revalidated, and updates must be picked up once again by the new primary, so the client should not be capricious about requesting a new primary.

When a switch is made, both the new primary and the client will learn about it: the new primary's first job is to inform the client of its identity. In order to handle primary changeover gracefully without resorting to explicit disconnection messages, old primaries *figure out* that their services are no longer needed. This is straightforward because the client always knows the identity of its current primary. Whenever a putative primary goes to a client to make a pickup, it first verifies that it is still the primary server for that client. If the client is now being serviced by a new primary, the old one goes away and leaves the client alone. For example, suppose that client C has moved away from primary P1, and has called the matchmaker to obtain a new primary P2. During P1's next pickup, it learns that P2 is the new primary. P1 immediately stops making pickups from C.

It is tempting to blur the line between clients and primaries: why not run the primary server code on the client machine? This would reduce the number of messages required during an initial `strict_read()`. Unfortunately, clients are not trusted, so we cannot allow them to access secondaries directly. A malicious client could invalidate our consistency guarantees by ignoring our protocols and writing inconsistent values to secondaries.

2.3 Filesystem-Secondary Attachment

Primary servers are used principally as intermediaries between clients and secondaries. Primaries do not function as replication sites *per se*. Instead, we use groups of secondary servers to replicate file systems. For every file system, there is a corresponding group of secondary servers that handles the replication of files in that file system. Because we support incomplete replication, it is not necessary that every file be replicated on every server, however.

In practice, a server can simultaneously act as a primary and a secondary. A group of machines may be used both to replicate files and to act as primary servers for clients. Logically, however, it is necessary to maintain the functional distinction between primaries and secondaries.

A primary server learns about filesystem-to-secondary mappings by calling a *mapping server* specified by the client. Ordinarily, this `map()` function will be called only when a file system is mounted. This is a comparatively infrequent operation, so scalability should not be adversely affected.

We assume that secondary server mappings will rarely be changed, but we do provide a modification method based on Gray's idea of *leases* [5]. Mappings returned by `map()` are valid for only a limited period of time: relatively long (say, 10 to 30 minutes), but decidedly non-infinite. At the end of the lease period, `map()` must be called again to revalidate the mapping.

To modify a filesystem-to-secondary mapping, we must wait until all leases on the mapping expire. At that point, we can update the mapping and let `map()` start handing out the new information. In order to prevent starvation while waiting for leases to expire,

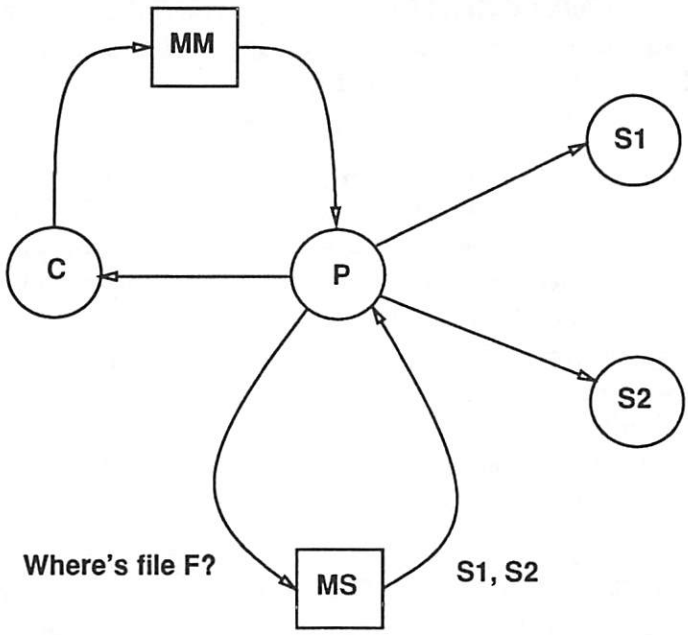


Figure 4: System Organization

the mapping server can hand out shorter leases after there has been a request to re-map. For example, if we know that all leases will expire in at most 30 minutes, we can still provide 29-minute leases on the same information.

The overall organization of our system is shown in Figure 4. There, C is the client, P is the primary, S1 and S2 are secondaries, MM is the matchmaker, and MS is the mapping server. Table 2 shows the state information that must be held by each component in the system, and whether the state needs to be held in non-volatile or volatile storage. In that table, the notation "File+timestamp cache?" refers to the fact that the client may store its updates (and their associated timestamp information) in either volatile or non-volatile storage; clients that store updates in non-volatile storage will not lose data in a crash.

2.4 Connection at a Distant Site

A desirable feature of our design is that a client can move to any location where network access is available and still continue to access the files at its "home site" in the same way that it would access them locally. For example, someone from New York who is visiting Berkeley would call the Berkeley matchmaker to obtain a local primary server. (We assume that some form of negotiation is made for the use of local resources.) The locally-chosen primary then calls the mapping server in New York to locate files that the client wishes to access. The client need remember only its own address and the address of the mapping server at its home site. These addresses need to be retained in non-volatile storage. Previous work on mobile internetworking by Ioannidis et al. [6] has demonstrated the feasibility of such a scheme from a networking perspective.

Under normal circumstances, we do not expect the primary server to play an im-

COMPONENT	NON-VOLATILE STORAGE	VOLATILE STORAGE
Client	Own address Mapping server address File+timestamp cache?	Current primary CTs held File+timestamp cache?
Primary	—	List of clients Filesystem-to-S mappings Files being propagated
Secondary	List of PCWs List of CTs	—
Matchmaker	List of primaries	—
Mapping server	Filesystem-to-S mappings	—

Table 2: State Held by Each Component

portant role as an intermediate cache. Muntz and Honeyman [9] report that intermediate caches experience a surprisingly low hit rate: less than 19% when the client itself has a 20 megabyte cache. Blaze and Alonso [2] achieve better results by using a dynamic, hierarchical caching scheme. In their model, however, clients are assumed to have minimally-shared hierarchies such as home directories and temporary files stored on local disks, which is not an assumption we wish to make in our system. When the primary is far from the secondaries, however, even a relatively low intermediate cache hit rate is a significant advantage for a client whose files would otherwise have to be shipped from a distant site.

3 Failure Recovery

A major advantage of our design is that it makes the algorithms for failure recovery extremely simple. In this section, we describe how crashes and partitions are dealt with.

3.1 Secondary Server Failure

Failure of a secondary server will not be noticed unless so many secondaries fail that an initial `strict_read()` can no longer guarantee consistency.

Recovery is easy: the recovering secondary is not required to do anything when it comes back up because an initial strict read will always contact enough secondaries to ensure correctness. However, finding and copying over the most recent versions of files is a good idea because it will increase the likelihood that `loose_read()` will return up-to-date values. The process of getting up to date can be done simultaneously with accepting updates — there need not be a synchronous "update phase." This is a consequence of the quorum requirement: there are no assumptions made about which secondaries have the most recent values.

3.2 Primary Server Failure

It is the client's responsibility to detect when its primary server has crashed. When its requests go unanswered, the client asks the matchmaker for a new primary. By letting the client drive the process, and by recognizing that the new primary is not required to have

up-to-date versions of any files, we avoid the complexity of a traditional election scheme [3].

Furthermore, a primary can crash (or just drop out) at any time without causing updates to be lost. Because a client is required to hold an update in its cache until receiving a purge notice — at which point the update is replicated on N or more secondaries — it is impossible for a primary to hold the only copy of an update. The newly-chosen primary will automatically restart the update propagation process for any updates remaining in the client's cache. No special logic is required; the new primary treats this case in exactly the same way as regular update propagation. The only minor inconvenience is that a client's currency tokens must be revalidated immediately after attaching to a new primary; the client-primary-secondary hierarchy upon which CTs rely may have been temporarily disrupted by the crash.

Recovery is trivial, again because all the state the primary held can be regenerated elsewhere. A primary that has come back up simply waits for the matchmaker to pair it up with clients. The recovering primary does not retain any state pertaining to its interactions with previous clients.

3.3 Client Failure

As in any write-back caching system, it is possible to lose data held at the client when the client crashes. Due to the use of asynchronous update propagation, our window of danger is somewhat wider than in other systems. However, if the client's cache is non-volatile, updates need not be lost, even if they are unavailable when a client is disconnected.

When a primary server is unable to make a pickup from a client, it assumes that the client has crashed and so it stops making pickups. If the client has not crashed, no harm is done: the client will eventually ask the matchmaker for a new primary. This is exactly what the client will do during recovery if it actually has crashed.

3.4 Reaction to Partition

During a network partition, an initial `strict_read()` cannot be performed in a partition containing fewer than T-N+1 or a majority of secondaries, whichever is greater. T-N+1 secondaries are needed to locate the most recent version of the file, while a majority is needed to read and update the PCW and CT lists.

Similarly, when fewer than N secondaries are reachable, the `write()` operation will block if the client's cache fills due to the primary's inability to propagate updates to a sufficient number of secondaries.

3.5 Resolution of Conflicting Updates

In general, the most recent version of a file, as determined by timestamp, is the one that will be retained. Timestamps are assigned by clients, but since clients are not trusted, timestamps are validated by primaries during each pickup. If the client-assigned timestamp falls outside the range of plausibility, the primary assigns its own timestamp and informs the client of the correct time for future use.

Even when `strict_read()` is used exclusively, conflicts can still occur. For example, two clients may strictly read the same file, and then make conflicting updates into their

caches. Ideally, if client A updates a file while client B is modifying its cached copy, client B should eventually learn that its updates were based on stale data. Client B's updates should not be written to secondaries without explicit authorization from B when such a conflict occurs. Conflict detection is particularly important after a network partition heals: we do not want updates to be overwritten with inconsistent versions that happen to have later timestamps.

We detect conflicts in our system through the use of timestamps augmented with a unique identifier (such as an IP address) that indicates which client last modified the replica in question. [2] After a read or a successful write (signified by receipt of a purge notice), the timestamp/client pair associated with that version of the file is retained by the client. On future writes, if the service determines that a newer version *written by a different client* has superseded the cached copy, a message concerning the conflict is sent to the client currently attempting to write.

There is no guarantee that the conflict message will arrive at the client, but this is not a problem. In the worst case, another message will be sent during the next round of pickup and propagation.

Because of our asynchronous update propagation, a client must maintain *two* timestamp/client pairs for each file in its cache: a *base* timestamp, indicating the version upon which updates are based, and an *eventual* timestamp, which is the ID that will be assigned to the current version after propagation is complete. If a client is not a PCW for a given file, the base and eventual timestamps are identical. The eventual timestamp is given out to clients who read a file from a PCW's cache during an initial strict read: this is the version on which other clients' updates will be based, whether or not that version has been fully propagated.

The client ID is necessary to detect conflicts because no order is mandated for update propagation. Without knowing which client is responsible for an update, it would not be possible to tell the difference between out-of-order updates from a single client (which is fine), and out-of-order conflicting updates from multiple clients (which must be flagged).

It is possible for an irresponsible client to *loosely* read a file, modify it, and write it back. However, one of the central themes of our work is that the cost of consistency should be borne by the users who demand it. If a client with write access to a file refuses to use `strict_read()` before modifying the file, that's the client's problem. A user who refuses to bear the cost of consistency is not guaranteed to receive it. Since write access is (presumably) granted only to clients who are willing to take responsibility for modifying a file properly, conflicts resulting from this problem should be very rare in practice.

3.6 Accommodation of Wandering Users

Finally, there is a case — unique to mobile operation — that we call the *wandering user*. Suppose a client strictly reads a file just before a network partition occurs. This person

[2] Although version vectors are a more powerful conflict detection mechanism than timestamps, the added power would not provide a significant advantage in our system. Version vectors are useful for detecting conflicts in an *optimistic* system such as Coda [12], which always returns the best value it can find. In contrast, our `strict_read()` operation will not return any value unless a certain level of consistency can be assured, thereby preventing many potential conflicts from occurring. Since the use of version vectors requires synchronously executing a protocol to exchange and update vectors, we ruled out their use.

then makes several updates to the file, which are propagated to all reachable secondaries. Before the partition heals, the client wanders into a different partition and makes more updates to the same file, which are propagated to a different subset of secondaries.

This appears to be potentially chaotic, but actually works out quite well. The client can make as many updates as desired during the partition because unavailable replicas do not cause write operations to block. When the partition heals, timestamps make it easy to decide which versions the secondaries should retain. If several wandering users make updates to the same file during a partition, the conflict detection scheme described above will sort out the divergent versions.

3.7 Semantics

Having reviewed the operation of our design during both normal and failure conditions, we now summarize the semantics of the three operations.

- The value returned by `loose_read()` is unpredictable, as is the fate of a write following a loose read.
- When a `write()` operation following a strict read returns, the client is assured that, if it remains reachable long enough, then its update will be stored at N secondaries, and — if it has not been superseded by a more recent conflicting update — will be installed as the latest value.
- In the presence of an arbitrary combination of writes by both loose readers and strict readers, the value returned by `strict_read()` is unpredictable.
- In the presence of only writes and strict reads, a strict read operation will return the value which is the latest among:

  1. The latest value written by a strict reader and present at any secondary server. The propagation of values to secondaries is subject to variation depending on the pickup schedules of primaries.
  2. The latest value in the cache of any reachable client which is a PCW.

  When there are no failures, `strict_read()` returns the "latest" value; when there are failures, this fact can be detected and returned to the user.

4 Conclusion

We have presented the details of a variable-consistency file service, emphasizing the dynamic relationships of clients, servers, and files. In addition, we have described and analyzed the recovery procedures for various types of failures, including the detection of conflicting updates.

Although mobile operation served as the initial motivation of our thinking, we think our proposal contains profound advantages for "regular" operation as well. The major novel features of our design are:

1. An interface in which the required consistency must be declared by the reader.

2. Currency tokens, which can drastically reduce the cost of strict reads after the overhead of the initial strict read.

3. An asynchronous update propagation scheme driven by the primary server, in which client cache space is traded for a low-latency, reliable write operation.

The extra information provided to the service through the interface allows the implementation of both `write()` and `loose_read()` to be made completely lazy, and hence fast and scalable. Another major benefit is considerable simplification of the internal algorithms, especially those for recovery; very little state need be kept in non-volatile storage. In essence, our design heaps the entire burden of implementing consistency onto the initial strict read. Thanks to the currency token — which persists until explicitly revoked, independent of cache occupancy — an initial strict read need be done very infrequently.

In our design a "primary server" is not a true replication site, but rather a bigger cache for reading and a coordinator for the asynchronous update propagation. By using the primary in this way, a client can be ignorant about the actual location of file replication sites. More important, though, is the elimination of blocking during typical write operations, courtesy of the beneficent primary.

We see two major drawbacks in this design: the need to program according to a new interface (although this can be mitigated by using one of the generic read calls suggested in Section 2.1), and the mediocre performance of the initial strict read. Evaluation of this file service is a usability question; to clear it up, we are at work on a prototype implementation.

5 Acknowledgements

This work was supported in part by National Science Foundation grant CDA-9022123, IBM Corporation, and the Center for Telecommunications Research, an NSF Engineering Research Center funded by grant ECD-88-11111.

References

[1] M. Baker et al. Measurements of a Distributed File System. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, pages 198–212, October 1991.

[2] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proc. Twelfth Intl. Conf. on Distributed Computing Systems*, June 1992.

[3] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. Computers*, C-31(1):48–59, January 1982.

[4] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. Seventh ACM Symp. on Operating System Principles*, pages 150–162, December 1979.

[5] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. Twelfth ACM Symp. on Operating System Principles*, pages 202–210, December 1989.

[6] J. Ioannidis, D. Duchamp, and G. Q. Maguire, Jr. IP-based Protocols for Mobile Internetworking. In *Proc. SIGCOMM '91*, pages 235–245, September 1991.

[7] T. Mann, A. Hisgen, and G. Swart. An Algorithm for Data Replication. Technical Report 46, Digital Systems Research Center, June 1989.

[8] D. L. Mills. Network Time Protocol (Version 2) Specification and Implementation. Network Working Group RFC 1119, September 1989.

[9] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems. Technical Report CITI TR 91-3, University of Michigan, August 1991.

[10] J. Ousterhout and F. Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[11] J.-F. Paris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *Proc. Sixth Intl. Conf. on Distributed Computing Systems*, pages 606–612, 1986.

[12] M. Satyanarayanan et al. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Computers*, 39(4):447–459, April 1990.

[13] A. Siegel, K. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. In *Proc. 1990 Summer USENIX Technical Conf.*, pages 51–62. USENIX, June 1990.

[14] C. Tait and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 190–197, IEEE, December 1991.

Using SCSI to Control Almost Anything *

Bruce Robertson
Hundred Acre Consulting
Reno, NV
+1-702-329-9333
bruce@pooh.com

1 Introduction

Whenever an intelligent, programmable device is developed, the question of how to communicate with it invariably arises. There are several options. Serial and parallel ports are simple in hardware and software, but suffer from being slow. Ethernet is (usually) fast, but requires some fairly expensive hardware, and complex network protocols and configurations.

By comparison, the SCSI bus is fast, and can be implemented with a single inexpensive chip and some resistors. It's true that the complete SCSI protocol is quite complex, but a very simple implementation often suffices. A SCSI device is also very easy to configure — all that needs to be specified is a SCSI ID, which is an integer between 0 and 7. Also, unlike Ethernet, very little of the SCSI bus bandwidth is typically used, even when a local hard disk is attached. These attributes make the SCSI bus an ideal interface for intelligent embedded systems.

SCSI has evolved from a fairly simple interface that controlled only disks into a detailed, general interface for controlling and communicating with almost any intelligent system. The SCSI standard includes details on how to communicate with several device types, including disks and tape drives, CD ROMs, communications devices, and even other hosts. Unfortunately, the SCSI drivers found on most modern UNIX systems only have support for disk and tape drives. Support for communications devices is almost always omitted. To avoid maintenance and porting nightmares, it is always highly desirable to avoid requiring end users to update their host software with a custom SCSI implementation.

The solution to this problem is to implement all custom SCSI devices using the disk or tape drive model. For many applications, the tape drive model makes sense, but this paper concentrates on the disk drive model.

2 Disk Device Model

As mentioned above, it is very important to avoid requiring changes to the host SCSI implementation. To do this, we make the device look like a normal SCSI disk drive. The only requirement of the host is that it can access the bare disk drive from a user program, without requiring a file system to be present.

The "blocks" of the disk drive model are divided into three regions. Since most workstations require a label block to appear at the beginning of a disk, the first 32 blocks are mapped directly to a 32 block section of memory, and disk reads and writes transfer data to and from this section of memory. This is essentially the same thing as a RAM disk. The label area should be implemented in non-volatile RAM, so that any labels the host writes are retained while the power is off.

The next several blocks are termed the "special" blocks, and usually don't behave as expected when read from or written to. These blocks are used to implement the various control and status

*This research was commissioned by Gradient Technology, 95 B Connecticut Drive, Burlington, NJ

functions required to communicate with the device. Typical special block functions include resetting the device, starting a program executing on the device's local processor, transferring data between the host workstation and a program running on the device, and commands for implementing a remote file system (described below).

The remaining blocks are mapped directly to the RAM on the device, allowing the host to access the RAM in a fashion similar to DMA.

3 Remote File System

Programs running on the target device can use the SCSI bus to access a local hard disk or floppy drive. However, many applications don't justify the cost of a local disk. Also, since the target device has only one SCSI bus, it is behaving as a SCSI Initiator to its local devices, at the same time that it is a SCSI Target (many workstations don't tolerate other Initiators on the same SCSI bus). In these situations, it is still desirable for the target device to be able to run programs and access data. For this reason, we implemented a stateless remote file system in the spirit of NFS.

The remote file system is implemented using a set of the "special" blocks. One special block is used by a "server" program running on the host to poll for NFS-like requests from the remote device. Others are used to send back data and status information in response to the requests. The server reads its special polling block several times a second, looking for requests.

Just like in NFS, files are referred to using "handles", which are opaque to the firmware on the target device. In the current server implementation, handles are simply UNIX pathnames. For simplicity, the "stale handle" problem is ignored. In most applications, the files that are being accessed by the target device belong entirely to it, and are not being modified by the host. As long as this is true, the "stale handle" problem does not arise.

4 Problems

As mentioned above, programs that wish to receive information from the target device must poll. When no information is waiting for the host program, reading a polling special block simply results in a block of all zeros.

A much more efficient method would be to use the disconnection feature of the SCSI protocol. When a read of a polling special block is done, the target would disconnect from the SCSI bus, and then reconnect to the host when there was something useful to transfer. Unfortunately, the host operating system assumes that it is talking to a real disk device, and typically times out the disconnected request within a few seconds. This causes trouble.

A possible solution to this problem is to have the target device periodically reconnect to the host, and then disconnect again without transferring any data — sort of a "reverse" polling. Unfortunately, while this does not violate the rules of the SCSI protocol, we can't assume that arbitrary SCSI implementations will handle this properly.

Another related problem occurs when the target device crashes, or is powered off. If a program on the host is trying to poll the device when the device isn't listening, it will typically lock up the SCSI bus for long periods of time, and send annoying messages to the host console. Some workstations have even been seen to crash when this happens. Again, there is no universal solution to this problem, other than make sure that the target device is as reliable as possible.

5 Conclusions

This project has demonstrated that it is possible to use the SCSI bus as an efficient means of communicating with and controlling an intelligent peripheral device. The method used can be extended to be a general purpose networking mechanism, as shown by the remote file system implementation. The goal of not altering the host operating system software was met.

Issues in BBFS, a Broadband Filesystem

Bruce K. Hillyer and Bethany S. Robinson
AT&T Bell Laboratories
Crawfords Corner Rd., Holmdel, NJ 07733
bruce@vax135.att.com bsr@vax135.att.com

Abstract

Advancements in networks and computers are fostering the development of new services and applications, but the data storage subsystems are an increasingly severe impediment. BBFS is a research effort to examine ways to accommodate the storage needs of communication- and computation-intensive applications. We are challenged by real-time constraints on data such as multimedia video and audio, by gigabit rates, and by enormous data volumes. To meet application performance and semantic requirements, the BBFS model incorporates distribution, parallelism, and extensibility in the filesystem application interface and behavior set.

1. Extensible Filesystem Framework

For certain classes of new applications, the limitations of traditional UNIX® filesystems are intolerable. For instance, various applications in our lab require real-time response, continuous data streaming, application-specific semantics, and high bandwidth and storage capacity. Although known mechanisms would suffice for many of these applications, conventional filesystems cannot readily incorporate these mechanisms. Moreover, we despair of finding a new base set of I/O primitives that satisfy all filesystem client requirements while meeting the twin tests of high performance and simplicity of usage. For BBFS we want an expanding set of highly efficient filesystem behaviors, accessed through a familiar and stable application programming interface. We place application-specific functionality in the filesystem for performance, and encapsulate the new functionality to make it straightforward to use.

BBFS provides a framework to incorporate a wide variety of mechanisms, using extensibility in the application interface to the filesystem, and extensibility in the filesystem behavior set. The application interface is similar to the UNIX filesystem interface (`open`, `close`, `read`, `write`, `seek`, `ioctl`), but each call accepts an additional argument containing *hints*. This argument is a list of attribute-value pairs. Hints are a simple, general, and extensible way to convey application requirements and optimization information to the filesystem code.

To incorporate new behaviors in the filesystem, BBFS provides three approaches: typed files, embedded trusted code, and active files. *Typed files* encapsulate new structures and behaviors to make them generally available to applications. For instance, the striped file type transparently accommodates application I/O at a greater data rate than a single disk bandwidth. *Code extensions* can be attached to the filesystem to implement new behaviors activated in response to hints, that for reasons of performance and control, may need to be closely coupled with filesystem mechanisms. For instance, operations on continuous multimedia data streams need to interwork with the filesystem scheduler and buffer management to obtain end-to-end performance. Only trusted code extensions are linked into BBFS server modules: we do not have safe methods to insert arbitrary application code into the system. *Active files* are a weak form of persistent object, containing data and procedures: procedure bindings catch calls to filesystem primitives such as `open`, `read`, and `write`.

A wide variety of capabilities can be accommodated in this framework. Examples of application-specific processing include compression, encryption, access control, data reduction, index maintenance and use, and application-dependent consistency control. Performance

® UNIX is a registered trademark of Unix Systems Laboratories, Inc.

management extensions could involve prefetching and buffering policies, space management, and real-time scheduling of transfers. Ease of use issues are exemplified by multimedia video and audio: an application may display multimedia simply by opening the file with hints for continuous transfer, appropriate bandwidth and jitter values, and redirection to the framebuffer.

The naming issue raises an interesting conflict. We want location-transparent naming so that a user or application can use the same name for a file, independent of which nodes in the distributed system are used for access and data storage. On the other hand we want custom namespaces per user or process so that, for example, generic program names are bound to machine-dependent executable files. BBFS seeks to accommodate both the global and per-process views of naming by a hybrid scheme. A file is unambiguously named by three fields in the low-level ID: `creation volume`, `unique id` within that volume, and `version number`. Within a physical volume, a UNIX-like string pathname can be bound to the `(unique id, version)` pair. Thus the global transparent name for a file is a volume ID prefixed to a pathname within that volume, and the global name is invariant with respect to the particular machine or port where the volume is connected.[1] To form a process-specific custom namespace, `mount` and `unionmount` operations map string pathname prefixes to prefixes of global transparent pathnames. To resolve a filename, the BBFS application interface library searches the local mount table for the longest matching prefix, substitutes the corresponding global filename prefix (the first component is a volume ID), and probes a cache or queries the volume database manager to find the server having the volume. Users need not know numeric volume IDs even to perform mount operations: the volume database query mechanism can look up a volume ID given attributes that describe the volume.

2. Status and Issues

The BBFS design is intended to support both uniprocessor and multicomputer distributed implementations. The first implementation runs on the HPC multicomputer under the VORX distributed operating system environment.[2] HPC/VORX, operational in 1988, forms a high-bandwidth low-latency message passing system with switched communication at 113 Mb/s among 100 processor nodes and workstations distributed over a 30 mile network. The BBFS implementation on HPC has supported clients since 6/91. One research application plays monochrome video and audio from BBFS using software disk striping and synchronization. Ferret, a semi-production information retrieval application, stores images of AT&T archival photographs and 20,000 Bell Labs internal memoranda in BBFS. Ferret displays 20 pages per second on an experimental HPC workstation, and serves 3-4 pages per second to any of over 500 workstations on the AT&T internal network. As of 3/92, BBFS mechanisms for automatic space allocation, resolution of low-level IDs, and caching of filesystem datastructures are operating correctly in a test harness. We are currently implementing a user-level thread package for filesystem and application-specific code, and are designing the real-time scheduler and a high-performance namespace manager.

We have many future issues to address. One is the volume database. Should it be a relational database management system? What should it store? Another involves implementing a reasonably broad set of hints and file types, to make BBFS immediately useful and to support evaluation of this approach to extensibility. Do hints and typed files work simply and cleanly, or are there innumerable bad interactions? We need sensible ways to incorporate user-specified threads into a real-time filesystem, and ways to cope with the untrustworthiness of trusted code. Also, can we still meet real-time and performance goals if we reimplement this architecture in a UNIX environment (e.g., using raw I/O, filesystem processes locked in memory at high priority, and buffers locked into shared physical memory)?

---

1. When a volume (i.e., optical disk cartridge or magnetic disk spindle) becomes accessible to BBFS, the device driver notifies the volume database manager of the binding between volume ID and physical address.

2. R. D. Gaglianello, B. S. Robinson, T. L. Lindstrom, E. E. Sampieri, "HPC/VORX: a local area multicomputer system", *Proc. 9th Intl. Conf. Distributed Computing Systems*, Newport Beach, CA, June 5-9, 1989, pp. 542-549.

The Processor File System in UNIX® SVR4.2

Ashok V. Nadkarni

UNIX System Laboratories
190 River Road, Summit, NJ 07901

INTRODUCTION

With the popularity of multiprocessor systems, there is a need for system administrative utilities that enable processors to be taken off-line and on-line. This feature is especially useful in fault tolerant systems for system maintenance. Additionally, users have a need to know the characteristics of various processors on the system, since some applications may be sensitive to the characteristics of the processors that they run on. Thus, there is a need for a mechanism to query and manipulate the processors on a system, which constitute an important hardware resource. Such a mechanism can then be extended to query or manipulate other hardware and also software resources.

This paper describes such a mechanism via the use of a file system type called the *processor* file system. This file system represents the configured processors in the system as files and thus makes this service network extensible. There is much similarity with the */proc* file system[1] in SVR4.2.

LAYOUT OF THE *processor* FILE SYSTEM

In this design, each configured processor in the system is represented as a separate file in this file system. One design decision was how much information should be available in the file name itself. One extreme case would be where all the information about the processor that one can get from this file system would be available in the file name itself and the body of the file would contain nothing. Simply looking up a pathname would give all the information and there is no need to open and read a file. A major disadvantage of such a design is that the file names could get unmanageably long as more and more information gets added to a file. Alternatively, the names of processor files could simply be some identity numbers that uniquely identify the processors and all the information could reside in the body of the file. This is a cleaner model and uses the same file system structure as */proc*. Hence, the latter design was selected.

Figure 1 shows the layout of the *processor* file system. The *processor* directory is the root node of this file system. In SVR4.2, this file system is mounted on */system/processor* to facilitate future extensions under the */system* directory. Under the *processor* directory, there is a *ctl* (control) file plus one file for each configured processor. The *ctl* file will be described in more detail a little later.

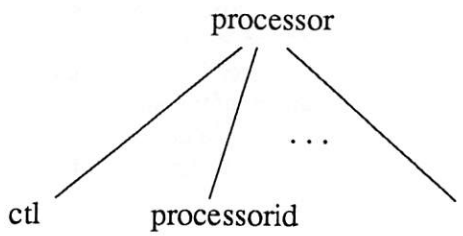


Figure 1 : Layout of the *processor* file system

Performing "*ls -l*" in the *processor* directory provides some interesting information as shown in Figure 2 below. A system with 3 processors is assumed in this example.

```
-r--r--r--   1   root   root   32   Mar 2   11:32   000
-r--r--r--   1   root   root   32   Mar 2   11:32   001
-r--r--r--   1   root   root   32   Mar 2   11:32   002
--w-------   1   root   root    0   Mar 2   11:32   ctl
```

Figure 2 : Output of "ls -l" in the *processor* directory

FORMAT OF PROCESSOR INFORMATION

Another design factor was the format of the information in the *processorid* files – whether it should be in ASCII format or binary. The binary format is adopted as it keeps the kernel code simpler and also avoids exposing the kernel code to issues such as internationalization.

CONTROLLING THE PROCESSORS' STATUS

The purpose of the *ctl* file is to allow privileged users (such as system administrators) to perform operations that are not allowed to be performed by regular users. Again, a design issue was whether to achieve this by performing *ioctl* operations on the *processorid* files. *ioctl* operations lack determinism in I/O direction which makes it difficult to separate cleanly the client/server interactions. Also we wanted to adopt a file system structure similar to the */proc* architecture in SVR4.2. Therefore it was decided not to support *ioctl* operations. Instead, a write-only *ctl* file is provided, that one can open and write one or more *messages* into the file. A *message* consists of a command followed by one or more operands (and is therefore extensible). Presently, the only commands supported are *online* and *offline* with a single *processorid* as the operand. Multiple *messages* can be combined into a single *write* system call.

STRUCTURE OF *processorid* FILE

Each *processorid* file is a read-only binary file. It contains the following information : processor status (i.e., whether it is on-line or off-line), the type of microprocessor used (i386/i486, etc.), the clock speed in MHz, the cache size in Kilobytes, whether it has a floating point unit (FPU), whether it has one or more drivers bound to it and the last time the processor state was changed (i.e., since when the processor has been on-line/off-line). All members of the structure are integers, except the last, which is of type *timestruc_t*.

IMPLEMENTATION

The *processor* file system fits into the Virtual File System (VFS) architecture[2] of SVR4.2 as one of the file system types. It is implemented as a pseudo file system, i.e., there is no backing store and all the information is maintained in main memory. The vnodes are generated on the fly, only when there is a reference to them. A vnode is generated during the *lookup* operation and deleted when the last reference to the file is released. As the total number of vnodes is quite small, no caching of free vnodes is performed. The algorithms used for vnode allocation and deallocation eliminates all race conditions by using appropriate mutual exclusion locks.

POSSIBLE EXTENSIONS

The *processor* file system can be easily enhanced to a more comprehensive set of file systems that provide network extensible access to both hardware and software resources of any machine on the network. For example, on systems that support the notion of *processor sets* and *light weight process sets*, these entities can also be represented in the file system name space. Each configured *processor set* would appear as a file under such a file system and the file would contain information about the processor set membership, and to which *light weight process sets* this *processor set* is "attached". It is possible to represent other hardware resources such as controllers and disks in a manner similar to what is done in the *processor* file system.

CONCLUSION

This paper described the *processor* file system that provides a useful mechanism to query and manipulate the processors in a multiprocessor system in a network extensible manner. The file system is stable and is a part of SVR4.2. If any of the extensions mentioned above are implemented, that would make such a collection of file systems even more useful to the computing community.

REFERENCES

1. Roger Faulkner, Ron Gomes, The Process File System and Process Model in UNIX® System V, *Usenix Conference Proceedings*, January, 1991.

2. S. R. Kleiman, Vnodes: An Architecture for Multiple File System Types in Sun UNIX, *Usenix Conference Proceedings*, June, 1986.

Placing Replicated Data to Reduce Seek Delays*

Sedat Akyürek & Kenneth Salem - University Of Maryland
Department of Computer Science
University of Maryland, College Park, MD 20742
email : akyurek@cs.umd.edu, salem@cs.umd.edu

Introduction

Random access time is a major factor that degrades the performance of disks. As improvements in CPU and memory speeds continue to outpace improvements in disk speeds, the importance of reducing random access times increases [Bitton 87]. Typically, seek time constitutes nearly half of the random access time [Bitton 88]. Average seek time can be reduced significantly by replicating some data on the disk and spreading the replicas across the disk's surface. To satisfy a request for replicated data, a suitable replica, e.g. the one nearest the disk head, can be retrieved. If the replica is closer to the head than the original, the seek time for the request will be reduced.

A variety of issues, such as replica placement, replica selection and total replica volume, can effect the performance of a disk with replicated data. In this paper we focus on replica placement, i.e., where to place replicas so as to maximize the reduction in seek time. In the hope of producing practical techniques, we have concentrated on heuristic solutions, which we have evaluated using trace-driven simulations. However, our approach is guided by an analysis of a simplified version of the placement problem for which we are able to develop an optimal solution. The work described here is presented in full in [Akyurek 91].

Replica Placement

To gain some insight into the issue of replica placement, we begin by addressing the following question: Given that a block b residing on cylinder c is to be replicated once, on which cylinder should the replica be placed? The location of the replica should be chosen to minimize the expected seek distance over a series of requests for data.

In our analysis, we make several simplifying assumptions. First, successive requests for data are assumed to be independent and identically distributed over the available data blocks. Second, requests are assumed be to serviced in first-come, first-served order. Third, when a replicated block is requested, the copy nearest to the current location of the disk head is retrieved. (In case of a tie, the original copy is used.) For simplicity, we assume that there is sufficient free space on every cylinder to hold the replica of the block under analysis.

In [Akyurek 91], we develop an expression to find the optimal place to put the replica of the given block, in terms of the original position of the block and cylinder access probabilities. This analysis can be applied directly to place replicas by measuring or estimating cylinder access probabilities, selecting a block for replication, and using the developed expressions to determine the position of the replica. In general, we will wish to replicate more than one block. While it is possible to extend our analysis to determine the optimal location of replicas of n blocks, the analysis and the result become cumbersome very quickly. Instead, we place the replicas of n blocks by considering them one at a time, in some order. For each block, the formulae developed in our analysis are used to place the replica.

Direct application of our formulae for determining block placement may be unnecessarily costly. Using insights from the analysis and from our experiments, we also propose several simpler replica placement heuristics called "distance partitioning", "mass partitioning" and "weighted distance".

*This work was supported by National Science Foundation Grant No: CCR-8908898 and in part by CESDIS.

Under the "distance partitioning" heuristic, a block's replica is placed on the cylinder two-thirds of the distance between the original's cylinder and the furthest edge of the disk. The cylinder access probabilities are not used. The "mass partitioning" heuristic is similar to "distance partitioning", except that the probability mass function for cylinder requests is used to determine "distance". A block's replica is placed so that two-thirds of the probability mass between the original and the end of the disk lies between the replica and the original. As in the "distance partitioning" heuristic, the replica goes to the side of original copy that contains the most probability mass. The "weighted distance" heuristic places the replica on a cylinder such that the product of the distance to the original and the cylinder's access probability is maximized.

Simulations

We have used trace-driven simulations to determine the reduction in average seek distance that can be achieved using the techniques we have presented in the previous section.

Traces were gathered from workstations running an instrumented version of the SunOS 3.2 operating system kernel. Each trace covers a period of approximately 24 hours. Before the simulated service of block requests begins, the simulator replicates a M blocks, placing the replicas using one of the heuristics presented in the last section. The M most frequently referenced blocks are replicated, once each, in decreasing order of reference frequency.

By replicating a relatively small portion of the data on the disks we have traced, the mean seek distance and the mean seek time were reduced significantly. Direct application of our analysis yielded mean seek distance reductions of 50-60% overall and seek time reductions of 24-34%. These results were achieved by replicating less than 2% of all the blocks on the disk. Although the simpler replica placement heuristics were not as effective as the direct application of the analysis, they still reduced the mean seek distance and mean seek time considerably. The "distance partitioning" and "mass partitioning" heuristics exhibited very similar performance, while the "weighted distance" heuristic outperformed them in terms of seek time.

Conclusion

We have discussed replication as a means of reducing seek times. We focused the problem of replica placement. An analysis of a simple disk model suggested a heuristic for placing replicas. This heuristic and several other computationally simpler alternatives were evaluated using trace-driven simulations. Our experiments indicate that replicated data placed using these heuristics reduces average seek distance and time significantly.

Data replication introduces a variety of issues other than replica placement. These include the handling of update requests, the selection of data to be replicated, and degree of replication. Our focus on replica placement was intended to break a complex optimization problem into manageable pieces. Our current work addresses these additional issues, building on our evaluations of placement heuristics to produce a comprehensive strategy for data replication.

References

[Akyurek 91] Akyurek, Sedat, Kenneth Salem, "Placing Replicated Data to Reduce Seek Delays," Technical Report CS-TR-2746 (UMIACS-TR-91-121), Department of Computer Science, University of Maryland, College Park, August 1991.

[Bitton 87] Bitton, Dina, "Technology Trends in Mass-Storage Systems," Proceedings of the SIGMOD 1987 Annual Conference, San Francisco, California, 1987.

[Bitton 88] Bitton, Dina, Jim Gray, "Disk Shadowing," Proceedings of the 14th VLDB Conference, Los Angeles, California, 1988.

Issues in Massive-Scale Distributed File Systems

Matt Blaze
Department of Computer Science
Princeton University

Rafael Alonso
Matsushita Information Technology Lab
Princeton, NJ

Background

Distributed file systems (DFSs), such as NFS [1] and AFS [2], are widely accepted as a convenient mechanism for sharing and distributing files among small- and medium- size groups of computers. Although the benefits of file system semantics could extend equally well to larger systems consisting of many thousands of machines spread over a wide geographic area, current systems do not scale up well enough to make such systems practical. While a number of commercial and experimental systems do aim for various kinds of scale, there is no system that would support, say, 100,000 machines located on several continents, all mounting a common /usr/bin directory in which software may be added or changed from time to time. Such large-scale systems, to the extent that they exist at all, rely on file replication protocols (such as Unix's *ftp, rdist,* and even *mail)* that may be less convenient, more difficult to administer, or less reliable than file system semantics. As networked computing systems become more prevalent, the demand for highly scalable file systems that are flexible, transparent, and easy to administer can be expected to grow as well.

All DFSs in current use make use of some form of caching, both to decrease server load and to improve client performance. The cache management policy is perhaps the single most important factor affecting DFS scalability. Our work focuses on the design of highly scalable cache management policies.

Traditionally, caching is thought of primarily as a technique to improve client performance, with the success of a caching scheme measured by the percentage of file accesses served by the cache (the hit rate). Massive scale, however, requires us to consider caches as serving a different function - reducing the percentage of file accesses handled by the server (the miss rate). While these two measures seem at first to be equivalent, they reflect different performance issues. The hit rate influences client access time, while the miss rate affects server load. A massive scale system might actually be willing to suffer slightly degraded client access time in exchange for greatly reduced server load.

Conventional wisdom tells us that massive file system scalability is relatively easy to achieve for immutable files but much harder (and perhaps not worth bothering with) for general, read/write files. We disagree on both issues. Simply ignoring cache validation for read-only files does not automatically imply scale; client cache misses and new clients can still generate enough traffic in a large system to swamp a server. File access traces suggest that occasionally-written files are too important to ignore, but that much larger scale may be readily achieved if clients can share cache data and help propagate cache invalidation messages when files are written.

Cache Hierarchies

One way to achieve greater scale is to free the server from the need to process the bulk of cache miss traffic. This can be done by organizing clients into a hierarchy, such that clients higher in the hierarchy attempt to serve cache misses for clients below them out of their own caches. There is some evidence to suggest that a carefully designed hierarchical cache organization could greatly reduce server load. In traces of activity at DEC-SRC [5] and at Princeton [3], we found that client cache misses are very often already in another cache. In a simulation of various LRU cache sizes, between 55% and 70% of cache misses (when counting individual opens for read) were for files that were in at least one other client's cache.

According to one experiment, however, the benefits of cache hierarchies can be surprisingly small. Muntz and Honeyman [4] used the DEC-SRC trace data to simulate a two level cache hierarchy. All 112 clients interacted with an intermediate infinite cache server instead of the actual server. While this did cause a small reduction in server traffic, from the client's perspective the intermediate cache was rarely used, even when the client cache was small. Depending on the size of the client caches, the hit rate at the intermediate cache was between 7% and 70%, falling off very rapidly (to about 10%-20%) for even small

client cache sizes. So although there was a modest benefit to the server, the intermediate cache actually introduced a substantial delay for the clients, since most requests not satisfied by the client's own cache were not in the intermediate cache either, and had to be fetched from the file server anyway. A multilevel hierarchy of finite caches could be expected to perform even more poorly for the clients.

Dynamic Hierarchies

The problem with a static hierarchy is that it is static. While they effectively reduce server load for very popular files used in many places, they introduce needless layers between client and server for those files not already in an upstream cache. We believe *dynamic hierarchies* are a promising strategy for scalable filesystems.

In a dynamic hierarchy, clients communicate directly with the file server for lightly-shared files but build a hierarchy of client caches "on the fly" for more widely shared files. In [5], we used our workloads to conduct a trace-driven simulation of a simple dynamic scheme.

In our simplified dynamic hierarchy, the file server maintains a conventional AFS-style list of clients with cached copies of each file. The server and clients also maintain a small bound Δ on the number of clients it is willing to serve for each file. The first Δ readers of a particular file get copies of the file and a guarantee that they will be notified if the file changes, just as in a normal server-invalidate system. Subsequent readers, however, get only the list of Δ previous readers. The new client uses some criterion (such as proximity) to select one of these previous readers and attempts to obtain the file there. The previous reader either sends the file or its own list of past readers, depending on whether its Δ for that file has been reached yet. Observe that this forms a tree of maximum degree Δ, which may be different for each file. Cache misses propagate back up in the opposite direction. For writes, invalidation messages propagate down the hierarchy from the first readers of the file to the later readers. In a dynamic hierarchy each client acts as a "mini-file server" for other clients, agreeing to keep track of them and pass on invalidation messages as they are received. Each client must also maintain a (relatively small) cache of mappings from files to machines from which to obtain them, should a read miss in the file cache. Observe that in a dynamic hierarchy, the work of the file server (and each client) server is limited to handing the initial read from a client and the cache miss traffic from the first Δ readers. Other cache miss traffic is shifted to the other participating clients, whose work is similarly bounded.

We used the DEC-SRC trace to drive a simulation of various values of Δ and of various cache sizes. The results were quite encouraging; for small values of Δ (2 or 3), this yielded a 50% to 80% reduction in server traffic depending on the client cache size. There was a slight increase in overall network traffic, from 5% to 20% depending on client cache size. These results suggest that even a very simple dynamic hierarchy yields what is probably a reasonable tradeoff between server load and client response time.

Of course, there are a number of unresolved issues that would need to be settled in order use dynamic hierarchies to build a practical large-scale system. In particular, a hierarchal scheme makes client code more complex than a flat one, and introduces additional points of failure. In our paper [5], we outline a number of these issues (cache consistency, fault tolerance, security, etc.) in more detail. Load balencing is a particularly interesting issue, since a client high in the hierarchy for a file gets better service for its own cache misses but balance this against providing service to other clients. We are currently implementing and taking real-time measurements of a prototype dynamic hierarchy for file distribution, using an ftp-like protocol.

Support for this work was provided by a Research Initiation Grant from I.B.M.

References

[1] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., & Lyon, B. "Design and Implementation of the Sun Network File System." *Proc. USENIX,* Summer, 1985.

[2] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanaryanan, M., Sidebotham, R.N. & West, M.J. "Scale and Performance in Distributed File Systems." *ACM Trans. Comp Sys,* V.6, No. 1, 1988.

[3] Blaze, M. "NFS Tracing by Passive Network Monitoring," *Proc. Winter 1992 Usenix.*

[4] Muntz, D. & Honeyman, P. "Multi-Level Caching in Distributed File Systems", *Proc. Winter 1992 Usenix.*

[5] Blaze, M. & Alonso, R. "Dynamic Hierarchical Caching in Large-Scale Distributed File Systems," *CS-TR-353-91,* Department of Computer Science, Princeton, 1991. (To appear in *Proc. 12th Intl. Distributed Computing Systems Conference,* June 1992).

AFS 3.1: A Stacked Vnode Implementation

Michael T. Stolarchuk
mts@citi.umich.edu
Center for Information Technology Integration
The University of Michigan
Ann Arbor, Michigan

ABSTRACT

The AFS 3.1 Cache Manager is an example of a stacked vnode implementation. Unlike other examples of stacked vnode implementations, it exhibits significant degradation for reads and writes to locally cached files. Possible reasons for the performance degradation are discussed.

1. Introduction

We have been investigating the poor read/write performance for files locally cached on AFS 3.1 clients [Satya]. The AFS 3.1 Cache Manager stores files in the local client file system by using the vnode interface. The construction of the AFS 3.1 Cache Manager is similar to a caching module discussed in [Rosenthall], where a new vnode implementation is also discussed. Rosenthall describes several examples of stacked vnode systems, each providing some additional features based on existing services. From his examples, it seems straightforward to construct a high performance stacked vnode implementation. Rosenthall describes the performance of a prototype, though not specifically the performance of his read-write caching modules. In the prototype, there is no detectable degradation.

However, from observations of processor requirements, we believe it is difficult to construct a high performance stacked vnode implementation.

2. Measurement

The AFS 3.1 Cache Manager uses the Berkeley fast file system (UFS) to store its files. We compared the read times of the Berkeley fast file system (UFS) against the read times of locally cached AFS files. We determined the length of the code path of several components: AFS, UFS, and a memory copy. The results are summarized in Figure 1 and Table 1. The figure graphs time spent in milliseconds against different sized read requests, varying from 1 byte to 10K.

All tests were run on an IBM RT/PC running AOS (BSD 4.2). Memory copy times were generated by timing a bcopy routine call in user space.

3. Discussion

Users expect the read time of AFS 3.1's locally cached files to be similar to read time of files from the local file system. Instead, a process reading a locally cached AFS file runs at half the speed of the same process reading a file within the local file system. Some AFS 3.1 users have decided not to use AFS to store binaries due to the slow read time of locally cached files.

Building a fast stacked vnode implementation above UFS requires knowledge of the processing requirements of UFS, which are small. A stacked vnode implementation with 10% overhead (at 4K) needs to run at 10% of 1.025 milliseconds (the time to perform the 4K UFS read), or about the time for a 512 byte memory copy. The 512 byte copy represents about 128 memory loads, and 128 memory stores on the RT; the 10% overhead would represents a small number of executable C statements.

Creating a high performance stacked vnode implementation is hard. Performance characteristics of the underlying vnode implementation have to be well understood. The stacked vnode layer will use processor cycles, degrading performance when compared to the underlying service. Creating synergism between the stacked layer and the lower layer is possible, for example to better match block size requirements, but the stacked vnode layer's processor needs will still remain.

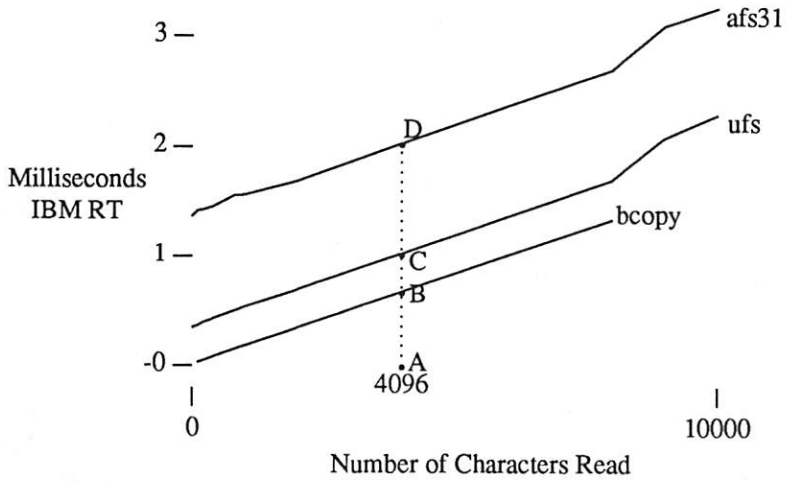


Figure 1. This figure graphs the performance of the AFS 3.1 (afs31), the Berkeley Fast File (ufs), and memory to memory copies (bcopy). Note at 4K reads, the overhead of AFS 31 (line segment CD) is as large as the time spent performing a similar read in UFS.

	IBM RT Line Segment Lengths				
Bytes	AC ufs	AB bcopy	BC overhead	AD afs31	CD overhead
100	**0.364**	**0.028**	0.336	**1.420**	1.056
1000	**0.536**	**0.182**	0.354	**1.567**	1.031
2000	**0.696**	**0.347**	0.349	**1.693**	0.997
4000	**1.025**	**0.675**	0.350	**2.035**	1.010
8000	**1.707**	**1.342**	0.365	**2.708**	1.001

Table 1. The table describes line segment lengths for the IBM RT performance graph in Figure 1. The numbers in bold are measured, the other values computed. All times are measured in milliseconds.

4. Conclusion

UFS uses very little processor resource to perform reads. A high performance stacked vnode implementation needs to be aware of the resource requirements of the underlying layers. A stacked vnode implementation above UFS can perform poorly due to its processing requirements.

5. References

Rosenthall. David Rosenthall, "Evolving the Vnode Interface", *Usenix Conference Proceedings*, pp. 107-117 (Summer 1990).

Satya. M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, and A. Z. Spector, "The ITC Distributed File System: Principles and Design,", *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (1985).

LOS ALAMOS HIGH-PERFORMANCE DATA SYSTEM

M. William Collins, Granville Chorn, Ronald Christman, Danny Cook, Lynn Jones, Kathleen Kelly, D. Lynn Kluegel, Christina Mercier, and Cheryl Ramsey
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Advances in massively parallel, large-memory computers and high-speed cooperative processing networks have created a high-performance computing environment that allows researchers to execute large-scale codes that generate massive amounts of data. A large problem will generate from tens of gigabytes up to several terabytes of data. These requirements are one to two orders of magnitude greater than what the best supercomputing data storage systems are now able to handle and will require a new generation of data storage systems. The Los Alamos High-Performance Data System (HPDS) is being developed to meet the very large data storage and data handling requirements of this high-performance computing environment. The HPDS will consist of fast, large-capacity storage devices that are directly connected to a high-speed network and managed by software distributed in UNIX workstations. The HPDS model is shown in Figure 1. Disk devices are used to meet high-speed and fast-access requirements while tape devices are used to meet high-speed and high-capacity requirements. Each storage device will have a dedicated workstation associated with it to provide storage and device management and to provide control for the data transfer. The file server component of the HPDS will implement user interface and file management capabilities that are distributed on multiple workstations. By connecting the disk and tape devices directly to a high-speed network based on the High-Performance Parallel Interface (HIPPI) standard of 100 megabytes per second and using UNIX workstations for the control, higher data transfer rates and reduced hardware costs are realized.

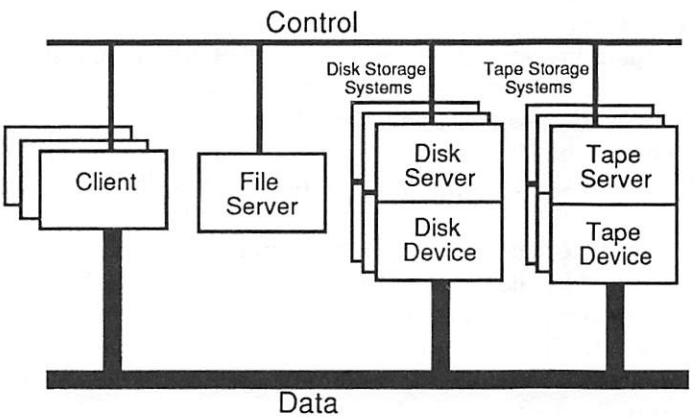


Figure 1. High-Performance Data System Model.

A prototype disk array storage system for the HPDS has been implemented by associating an IBM RISC/6000 workstation with an IBM 9570 HIPPI-attached disk array. All requests to store and retrieve data are made by client machines to the workstation, which then issues the read/write commands to the disk array through an Ethernet "command-only" port using TCP sockets. The read/write commands specify that the disk array is to transfer the data to/from the client machines using the HIPPI "data-only" port. The disk array will not accept commands on its HIPPI data-only port, so access can only be through the workstation, which provides data security and integrity. Device management and storage management capabilities for the disk array were implemented on the workstation.

As shown in Figure 2, this prototype disk array storage system was connected to the Los Alamos Advanced Computing Laboratory HIPPI network, which allowed the disk array storage system

to have HIPPI connections with a Thinking Machines CM-2, a CRAY Y-MP, an IBM 3090, and a high-resolution HIPPI frame buffer.

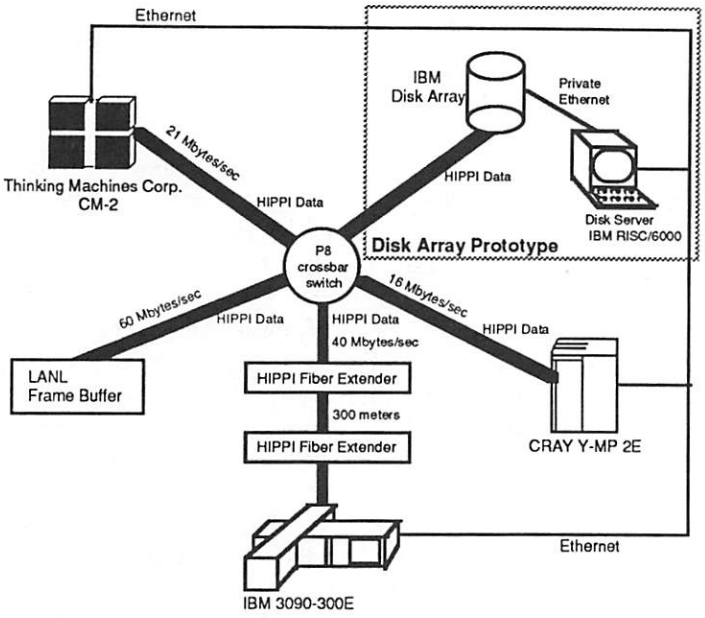


Figure 2. Prototype Disk Array Storage System.

A data transfer protocol was implemented for the disk array storage system and the client machines based on the separation of control and data where the control uses TCP socket connections and the "raw" data (data without headers) is transmitted over a HIPPI connection. This separation allows for the reliable delivery of the control messages while simultaneously allowing large blocks (megabytes) of data to be transferred over the HIPPI with minimal overhead. The protocol provides flow control, block-level retransmission, and timeouts. The data transfer can consist of whole files, parts of files, or appending to the end of a file.

Files were transferred between the disk array storage system and the Connection Machine (CM-2) DataVault at 21 megabytes per second (limited by the speed of the DataVault), the CRAY Y-MP disk at 16 megabytes per second (limited by the speed of the Cray disk), and the IBM 3090 expanded memory at 40 megabytes per second. To transmit visualization data from the disk array to the HIPPI frame buffer, the workstation issues a command to the disk array to write to the frame buffer. Files are transferred from the disk array to the frame buffer at 60 megabytes per second, which is approaching the maximum transfer rate of the IBM disk array. At these transfer rates, it is possible to transfer a two-gigabyte visualization file from the CM-2 or CRAY Y-MP to the disk array in less than two minutes and then to display the file on the frame buffer in 30 seconds.

The implementation of the prototype disk array storage system has demonstrated that UNIX workstations can be used to control the high-speed transmission of data over a HIPPI network between client machines and HIPPI-attached storage devices. The same approach will be used for HIPPI-attached tape devices when they become available. A UNIX workstation-based tape server will be associated with a HIPPI-attached tape device to implement a tape storage system.

The Coconut file system: utilizing tape-based robotic storage

Carl Staelin
Hewlett-Packard Laboratories

Robotic storage devices offer large storage capacities at low cost, but with large access times. While such systems have been common in supercomputer sites for years, they are now becoming affordable for many more customers. One problem with these devices is the lack of system software to seamlessly incorporate them into the storage hierarchy. The Coconut file system extends the Sprite log-structured file system (LFS) to incorporate such devices into the file system.This is joint work with John Kohl and Mike Stonebraker of the University of California at Berkeley.

1 Introduction

Robotic storage devices offer huge storage capacity with large seek times and fast sequential transfer rates, at very low cost per byte. Integrating these devices into the storage hierarchy will be the next challenge faced by file system designers. Coconut is being developed as part of the Sequoia 2000 Project, and it includes both conventional disk and robotic tertiary storage in a single file system.

Sprite LFS [Rosenblum91] was developed at the University of California at Berkeley by Mendel Rosenblum and John Ousterhout as part of the Sprite operating system. Its primary characteristic is that it is optimized for writing data, whereas most file systems are optimized for reading data. LFS divides the disk into 1MB segments and it writes data sequentially within each segment. The segments are threaded together to form a log, so recovery is simple. Disk space is reclaimed by copying valid data from dirty segments to the tail of the log, and then marking the segments clean.

2 Robotic storage devices

Robotic storage devices include at least three major media types: tape, optical read/write disks, and optical WORM disks. Robotic tape devices have several advantages: high capacity per medium (tape cartridge), on-device compression/decompression, and extremely low cost per byte. Modern cartridge tape drives can stream data between 0.25 and 50 Mbytes per second and store between 5 and 125Gbytes of data on a tape depending on the tape technology [Robinson91, Wood91]. Finally, the cost is significantly lower than disk technology (a 2.0Gbyte disk costs $1.60 per Mbyte).

Robot Device	*Medium*	*Media Capacity*	*Number of Media*	*Maximum Storage Capacity*	*Cost per Mbyte*
Metrum	Tape (VHS)	14.5GB	600	9TB	$0.001
Exabyte-120	Tape (8mm)	5.0GB	116	580GB	$0.17
Exabyte-10	Tape (8mm)	5.0GB	10	50GB	$0.20

3 Existing Systems

Tertiary storage devices have been around and in common use at supercomputer sites for several years. Alternative technologies for making use of robotic (tape/optical) devices include: file migration/archival storage, the Inversion file system, and the Plan-9 file system. File migration strategies copy whole files to and from the robotic storage [Smith81b]. However, it is inefficient for large files (perhaps larger than the available disk capacity) which are only partially accessed. The Inversion file system is built on top of Postgres. It assumes that the database storage manager can control the robotic device, and it implements file storage by having each file be a single object

within the file system relation. The Plan-9 file system uses a WORM optical jukebox with disk as a cache for the jukebox. It makes no special effort to cluster block transfers to/from the jukebox [Quinlan91].

4 Coconut file system

Log-structured file systems are a good match for archival storage, which is almost a write-only environment, since they are write-optimized. In order to optimize read performance, the cleaner, which flushes data from disk to tape, must cluster "related" data together on tape. Data is read from tape in 1Mbyte segments, so the cleaner tries to place closely related data within the same segment. We expect that data should be clustered in the following order: blocks within the same file, files within the same directory, and directories within the same directory sub-tree. Coconut will also prefetch segments from tertiary storage using read-ahead algorithms and hints left by the cleaner.

Coconut has a single block address space for both disk and tape blocks. Coconut allocates a fixed amount of space to each tape, but since tapes hold a variable amount of data, this number is set to be the maximum amount of data the tape is expected to hold. Block addresses consist of a segment number and an offset within the segment. The segment number determines both the device (or tape cartridge) and the offset within the device (or tape) of the segment. Coconut can easily handle device-based compression on tape since it can keep writing segments to tape until the drive returns an "end-of-tape" message, at which point the tape is marked full and the last (partially written) segment is re-written onto the next tape.

New data is written to disk-resident segments and is (eventually) migrated from disk to tape by the cleaner. The cleaner may create clean disk segments by copying data to the tail of either the active disk segment (this is how the cleaners in Sprite and 4.4BSD LFS work) or the active tape segment. Disk segments can be used to cache tape segments. Tape-resident blocks are accessed through the disk-resident segment cache, which is read-only since old data is never over-written.

Extending 4.4BSD LFS to Coconut requires relatively few modifications. The cleaner must be able to migrate blocks from disk to tape. The block-fetch routine within Coconut has to understand the difference between disk-resident and tape-resident blocks, and a segment cache must be added to cache segments from tape on disk.

5 Conclusion

In conclusion, I believe that robotic storage devices will become common in the near future, and that LFS on robotic tape storage devices is an attractive technology for transparently providing file system services on low-cost mass-storage.

6 References

[Quinlan91] Sean Quinlan. A cached WORM file system. *Software—Practice and Experience*, **21**(12):1289–99, December 1991.

[Robinson91] Harris Robinson. A mass storage subsystem using ANSI X3B6 ID-1 recorders. *Digest of papers, 11th IEEE Symposium on Mass Storage Systems* (Monterey, CA), pages 43–5, 7–10 October 1991.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. Association for Computing Machinery SIGOPS, 13 October 1991.

[Smith81b] Alan Jay Smith. Optimization of I/O systems by cache disks and file migration: a summary. *Performance evaluation*, **1**:249–62. North-Holland, Amsterdam, 1981.

[Wood91] Tracey G. Wood. A survey of DCRSi and D-2 Technology. *Digest of papers, 11th IEEE Symposium on Mass Storage Systems* (Monterey, CA), pages 46–50, 7–10 October 1991.

The Delta File System (ΔFS)

Cyril U. Orji *Jon A. Solworth*
University of Illinois at Chicago, Department of EECS (m/c 154)
Box 4348, Chicago, Illinois 60680

orji@parsys.eecs.uic.edu *solworth@parsys.eecs.uic.edu*
(312) 996-0955

Introduction

The Delta File System (ΔFS) is a Unix file system which uses logging of meta-data to achieve high performance I/O on small writes. Unlike other meta-logging systems, ΔFS uses a *threaded* log, representing the log as a linked list on disk. While this somewhat increases the time for recovery, it allows for extremely fine-grain synchronous operations. ΔFS also efficiently supports asynchronous operations [OS91]. Since ΔFS efficiently supports both synchronous and asynchronous writes, we shall say that it is *policy independent.*

Logging file systems enable consistent, ordered updates to the files to be maintained in the log, without requiring random seeks to update meta-data as in traditional Unix file systems, ex. [MJLF84]. Moreover, logging provides a mechanism for atomicity, eliminating the need for *fsck* [MK86] after non-media failures, significantly speeding up recovery time.

However, all logging file systems that we know of, for example, LFS [RO91b], Log files [FC87], and Episode [CAK+92], increase asynchrony to increase performance. Although more asynchronous semantics mean that more information is lost in the event of a system crash, this tradeoff is acceptable for many systems. However, it may not be acceptable for fault tolerant, distributed systems, or database applications. Moreover, it is incompatible with many existing applications, most notably NFS [San85].

In ΔFS the log is threaded as a linked list, and hence log blocks can reside anywhere on the disk — the head of the linked list is the tail of the log[1]. Like file blocks in traditional Unix file systems (ex. FFS), log blocks can be written anywhere there is an unused sector. This eliminates seeks (since there is almost always a free sector on the current cylinder) and almost as important, it eliminates most rotational delay[2] for meta information update.

When compared to other software techniques, threaded logs provide fine-grain synchronous support. Hardware techniques, such as NVRAMs can be used with contiguous logs — however, these are most appropriate for servers in which the extra cost, redundancy to prevent failure, reliability, and systems practices make them a much more effective solution than for workstations. For more details on the advantages of ΔFS, see [OS91][3].

NFS write rate

As described in the previous section, there are many applications which require synchronous semantics. In this section, we consider one such application, Sun's Network File System (NFS) and

[1] Recovery after system failure either requires a pointer which survives system failure to the last block in the log or requires log blocks to be written at preselected locations. From the head of the list, the pointers can be used to thread back as far as necessary in the log.

[2] For example, on a 10 surface disk, using 10% free space, there is an average of one free block per sector, resulting in some free block under a write head.

[3] This and other technical reports are available by anon. ftp to parsys.eecs.uic.edu: ftp/papers

size (kbytes)	FFS		ΔFS		
	time (ms)	mbytes/sec	time (ms)	mbytes/sec	improvement
4	41.67	0.096	5.34	0.748	7.8
8	55.56	0.144	7.41	1.080	7.5
16	58.82	0.272	12.66	1.264	4.7

Table 1: ΔFS and FFS performance on synchronous NFS transfers

examine its performance under both FFS and ΔFS.

NFS is a stateless protocol, which ensures that when a failed server is brought up, none of the information sent to the server is lost (except in the event of a disk media failure) [San85]. The protocol requires that every block written must be committed to safe memory, after which the client receives an acknowledge. This synchronous block write, in addition, requires an update either to the *inode* or *indirect block*. Hence each block requires two writes, limiting performance on the FFS. Although hardware solutions such as Prestoserve[4] have proven effective at using nonvolatile RAM (as safe memory), they add expense while introducing a new point of failure.

In Table 1, we examine the effect of large (multiblock) NFS writes from a single client. This table considers both FFS and ΔFS performance. A 1 ms network time is assumed after acknowledge. Using only volatile RAM, and hence forcing a log block to be written for every file block, 4K synchronous writes on ΔFS achieved 7.8 times the performance of FFS without any additional hardware. Sun uses a slightly larger block size of 8K, and the ΔFS speedup is only slightly diminished — 7.5 times FFS. As the block size increases, the speedup advantage slowly decreases (for example at block size of 64K, the advantage is reduced to 2.5 times that of FFS).

References

[CAK+92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, A. Mason, and R. Sidebotham. The Episode File System. In *Proceedings of the Winter USENIX Conference*, January 1992.

[FC87] R. Finlayson and D. Cheriton. Log Files: An Extended File Service Exploiting Write-Once Storage. In *Proceedings of the Symposium on Operating Systems Principles*, pages 139–148, Austin, Texas, November 1987.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2 No. 3:181–197, August 1984.

[MK86] M. McKusick and T. Kowalski. *Fsck—The UNIX File System Check Program.* UNIX System Manager's Manual, 4.3 BSD, Virtual VAX-11 Version, USENIX Association, Berkeley, CA, 1986.

[OS91] C. Orji and J. Solworth. The delta file system (ΔFS). Technical Report UIC-EECS-91-17, University of Illinois at Chicago, 1991.

[RO91a] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Symposium on Operating Systems Principles*, Pacific Grove, California, October 1991.

[RO91b] M. Rosenblum and J. Ousterhout. The LFS Storage Manager. *Proceedings of the Summer USENIX Conference*, pages 315 – 324, June 1991.

[San85] R. Sandberg. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130, Portland, Oregon, June 1985.

[4] SUN Microsystem Prestoserve Sales literature

An Intensional File System [1]

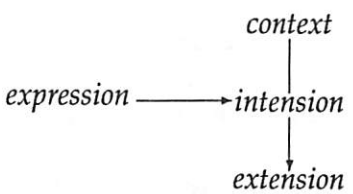
Paul R. Eggert
Twin Sun, Inc.
360 N Sepulveda Bl
El Segundo, CA 90245

D. Stott Parker
Computer Science Department
University of California
Los Angeles, CA 90024-1596

Today's file systems store files *extensionally*, as a sequence of data items like bytes or blocks. This scheme is the simplest and most efficient method for traditional computers where main memory can be assumed to be limited and disk-to-memory bandwidth adequate. However, these traditional assumptions are becoming obsolete: in more modern, distributed systems, file system clients typically have both cycles to burn and adequate memory for caches. In this new environment, it is often more efficient and convenient to store files *intensionally*, that is, as a description of how to compute the data items instead of as the data items themselves. In other words, an intensional file is a program that produces the corresponding extensional file as output.

This new opportunity meets a long-standing need for user-extensible file systems, a need that was recognized by earlier researchers. For example, this need is partially addressed by mechanisms provided by recent work in object-oriented file systems. However, most previously proposed mechanisms suffer two major drawbacks: they require kernel modifications or root privileges, and they are too complicated to be given to ordinary users. Much of this complexity stems from the ad hoc nature of the mechanisms' designs.

Intensional file systems provide a simple, easy-to-explain abstraction for implementing one file system on top of another, an abstraction that provides motivating design principles for file system designers, and that makes it easy for programmers to define new file system types. The underlying theory, intensional logic, has a rich history that provides new insights to file system design. Intensional logic ascribes meaning to a logical expression or name. Our work was inspired by Montague's intensional logic[1], which is particularly amenable to computational implementations, and has been widely used for natural language processing. In Montague's logic, an intension is the connection between a language expression and its worldly extension, as follows:



IFS0 is a simple example design that illustrates how this theory can be applied to file system names. IFS0 extends the Unix tree-structured file names by adding intensional paths, which are expressions evaluable with the shell. Intensional paths can evaluate to either file contents, or to names of other files. For example, the intensional path `(date)` evaluates to the current date, while the intensional path `$(sunp Xsun X)` evaluates to the file named by the output of the command `sunp Xsun X`.

[1]This research was supported by NSF grant IRI-8917907, and a State of California MICRO grant with Twin Sun, Inc.

We have designed, built and tested IFS0, using a system call interception scheme like that of COLA[2]. IFS0 is quite simple from the user's viewpoint. With IFS0 the user creates an intensional file using the ordinary Unix `ln` command, e.g.

```
$ ln -s '(date)' now.txt
$ cat now.txt
Sun Mar  8 20:06:31 PST 1992
$ awk '{print $4}' now.txt
20:07:01
```

To see whether a file is extensional, one invokes `ls -l` in the usual way. The ownership, length and date is that of the intension, not the extension. To see the extension, append the `-L` option; just as with symbolic links, it causes IFS0 to refer to the extension instead of the intension.

```
$ ls -l now.txt
lrwxrwxrwx  1 eggert          6 Mar  8 20:06 now.txt -> (date)
$ ls -lL now.txt
-r--------  0 eggert         29 Mar  8 20:14 now.txt
```

For a more practical example, consider the following make rules:

```
dist.tar :
        tar cf - *.h *.c >$@
dist.tar.Z : dist.tar
        compress <dist.tar >$@
clean ::
        rm -f dist.tar*
```

These can easily be replaced by intensional files that are always up-to-date, take far less space than their extensional counterparts, and never need cleaning, as follows:

```
ln -s '(tar cf - *.h *.c)' dist.tar
ln -s '(compress <dist.tar)' dist.tar.Z
```

IFS0 itself is surprisingly simple: its kernel is only 500 lines of C code. Even though no attempt has been made to tune the code, IFS0's performance is adequate for real use. Much of this is because there is no overhead for system calls like `read` and `write` that do not interpret path names.

We consider IFS0 a successful experiment that suggests several design considerations and future directions.

References

[1] W.H. Dowty, R.E. Wall and S. Peters, *An Introduction to Montague Semantics*, Boston: D. Reidel Publishing Company (1988).

[2] Eduardo Krell and Balachander Krishnamurthy, "COLA: Customized Overlaying", *Proc. Winter Usenix Conference*, San Francisco, 3–7 (January 1992).

A case for intelligent storage devices

Robert M. English
Hewlett-Packard Laboratories
renglish@hpl.hp.com

Abstract

The increasing power and decreasing cost of microprocessors has made it possible to incorporate a high degree of intelligence into peripheral storage devices, so that it is now feasible to move much of the functionality traditionally associated with a file system from the host computer into the peripheral, at once increasing system performance and improving modularity. The arguments for and against intelligent storage devices are examined, and a design is presented which allows the work of page allocation and data layout to be moved from the host computer to the peripheral.

Where is the best place to build a file system? Since the host computer has the fastest processor, the most information about program behavior and the greatest programmability, the answer to this. Moving file system functionality to peripheral devices loses important information and processing power, resulting in bad decisions, inflexible designs, and poorly performing systems.

And yet, when one looks at the world, one sees a very different picture. In many cases, file systems do not reside in host computers, but in dedicated file servers. Even in host computers, disks are sliced into pieces and used by different parts of the operating system in ways that interfere with one another. Logical volume managers such as the LVM from OSF insulate the file system from detailed knowledge of the disk—and improve performance by doing so. File system designs are difficult to change because of large installed bases. File system practice exhibits all of the behaviors normally attributed to intelligent peripherals.

Taking the arguments in more detail...

Processor performance

The first argument against intelligent peripherals is that the host processor is fastest, so that moving functions out to the storage device will slow the system down. As long as the peripheral processor is fast enough to keep up with the storage device that it serves, however, this argument is false. If it requires only 0.5 MIPS to keep up with a disk drive, then applying 50 MIPS will not improve performance unless the disk begins to run 100 times as fast, and by the time such a disk exists, inexpensive 50 MIPS imbedded processors will be available. For most file system actions, processor performance is no longer an issue.

Information

Moving intelligence from the host to the peripheral seems to imply a loss of information. The host system, after all, knows everything that could be known about application and system behavior, and a file system which could take advantage of that information should be able to outperform one located in a peripheral. Information and knowledge, however, are two separate things, and action yet a third. For the file system to act on the information available at the host, it must be able to interpret and understand a wide variety of data structures. Not only is this information difficult to extract, but the effort to do so leads to complicated, monolithic system structures that are very difficult to maintain. As a result, only a small part of the information available to the file system can actually be acted upon, and that information can easily be transmitted to an intelligent peripheral.

Flexibility

[Hennessy90] describes an intelligent disk controller from IBM that provided direct support for ISAM files, but came to hurt performance as the speed of the central CPU improved, and use this as an example of how architectures involving intelligent peripherals can fix design decisions past their point of utility. Unmentioned is the fact that host-based operating system or file system designs make the same types of decisions, with the same or greater implications for long-term system performance. MS-DOS systems, for example, are still crippled by decisions made for the first IBM-PC. Unix systems are crippled by decisions made 20 years ago. If these decisions had been imbedded transparently in plug-compatible peripherals, they would long since have been replaced with new generations of hardware.

I/O Performance

In a sense, all of these arguments are tangential. The real question is whether a system based on intelligent peripherals can outperform a conventional system. Recent work suggests that it can. [English92], for example, describes *Loge*, a peripheral that uses detailed information about disk characteristics and status to improve I/O performance without modifying file system code. Software such as OSF's Logical Volume Manager introduces a layer between the file system and the conventional device layer that prevents the file system from knowing the physical layout of the device, and often improves system performance by doing so. This type of gross remapping is clearly just as effective and just as feasible within an intelligent peripheral or disk array as within a host file system.

Modularity

Intelligent peripherals operating through standard interfaces are inherently more modular than software modules on the host computer. Different vendors can have different policies without their modules competing for system resources with each other and with applications. A small system with seven SCSI disks attached, for example, could quite easily have seven different file systems, optimized for seven different types of data. In a conventional system, these would require seven different copies of file system code and data structures, putting great pressure on the host computer. Or consider the difficulty of porting the Sprite Log-Based File System [Rosenblum91] to a small system. If memory were tight, the large (1MB) write buffers that LFS uses to achieve its performance would be difficult to justify. Similar techniques were applied within a peripheral would not compete for host resources.

Conclusion

While traditional system architectures offer the promise of flexibility and high performance, they fail to deliver on these promises. Decisions made in file system design make performance trade-offs that last for many years, and the difficulty of incorporating important information into file system decisions limits performance. The modular nature of intelligent peripherals allows them to be introduced into systems without modifying host software, lowering barriers to innovation and increasing performance

References

[English92] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *Usenix Technical Conference* (San Francisco, Winter '92), pages 237–51. Usenix, 20–24 January 1992.

[Hennessy90] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, Incorporated, San Mateo, CA, 1990.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. SIGOPS, 13 October 1991.

Multiprocessor File System Interfaces

David Kotz
Department of Math and Computer Science
Dartmouth College
Hanover, NH 03755-3551
David.Kotz@Dartmouth.edu

Introduction

MIMD multiprocessors are increasingly used for production supercomputing. Supercomputer applications often have tremendous file I/O requirements. Although newer I/O subsystems, which attach multiple disks to the multiprocessor, permit parallel file access, file system software often has insufficient support for parallel access to the parallel disks, which is necessary for scalable performance. Most existing multiprocessor file systems are based on the conventional file system interface (which has operations like *open, close, read, write,* and *seek*). Although this provides the familiar file abstraction, it is difficult to use for parallel access to a file. Scalable applications must cooperate to read or write a file in parallel.

We propose an extension to the conventional interface, that supports the most common parallel access patterns, hides the details of the underlying parallel disk structure, and is implementable on both uniprocessors and multiprocessors. It also supports the conventional interface for programs ported from other systems, programmers who do not require the expressive power of the extended interface, and access via a standard network file system.

We concentrate on scientific workloads, which on uniprocessors have large, sequentially-accessed files. Parallel file systems and the applications that use them are not sufficiently mature for us to know what access patterns might be typical, but we expect to still see sequential access either *locally*, within the access pattern of each process, or *globally*, in the combined accesses of all processes cooperating to access a file.

Extensions to the Conventional Interface

The Unix file system interface [5] is the typical conventional interface, supporting operations such as *open*, *create*, *close*, *read*, *write*, and *seek* on the file, considered to be an addressable sequence of bytes. Depending on the particular multiprocessor implementation of the Unix interface, there are many difficulties in using the interface to program a parallel file access pattern. We describe our extensions as solutions to these problems.

Sharing open files: Typically, each process must open the file independently, generating many *open* requests. This is both inconvenient and inefficient. We propose a *multiopen* operation, which opens the file for the entire parallel application when run from any process in the application.

Self-scheduled access: One globally sequential access pattern reads or writes the file in a self-scheduled order. The conventional interface requires the programmer to synchronize the processes, determine a file location for the next record, seek to that location, and perform the access. This is inconvenient and error-prone. We propose to support a both a *global* file pointer (providing a single shared file pointer for all processes, atomically updated on each access) as well as the traditional *local* file pointer (providing each process with an independent, local file pointer).

Segmented files: Consider the task of writing a large output file. One possibility is to write all of one process's data, followed by the next, and so forth. In parallel, each process seeks to the beginning of its segment of the file, and starts writing. This is difficult to do if the sizes of the segments are not known in advance. It is extremely awkward to extend a process's segment later. For these situations, we provide a new type of file called a *multifile*. A multifile is a single file with one directory entry, and contains a collection of subfiles, each of which is a separate sequence of bytes. A multifile is created by a parallel program with a certain number of subfiles, usually equal to the number of processes in the program. Each process writes its own subfile. Later, when the multifile is opened for reading, each process reads its own subfile.

Records: We support logical records, in addition to the traditional byte-stream abstraction. The record support can be combined with the global file pointer synchronization to provide atomic operations for reading and writing records.

Mapped File Pointers: To support access patterns other than self-scheduled and segmented, we allow the user to specify a *mapping* function for each file pointer, which maps the file pointer to a specific position. Some built-in functions (e.g., interleaved), are provided.

Coercion: With record files and multifiles, files are no longer simply a single sequence of bytes. To allow access by programs using the traditional interface, we provide automatic *coercion* of multifiles or record-oriented files into plain byte-oriented files. The interface provides the conventional abstraction without physically changing the file's organization.

Previous Work

One early implementation is the Intel Concurrent File System [4]. Crockett [1] outlines a multiprocessor file system design. The most exciting recent work is the new nCUBE file system [2] and the ELFS object-oriented interface [3].

References

[1] T. W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[2] E. DeBenedictus and J. M. del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, Apr. 1992. To appear.

[3] A. S. Grimshaw and J. Prem. High performance parallel file objects. In *Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

[4] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[5] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 6(2):1905–1930, July-August 1978.

Availability. The full version of this paper, Dartmouth technical report PCS-TR92-179, is available through anonymous ftp to `sunapee.dartmouth.edu` as `pub/CS-techreports/TR92-179.ps.Z`.

This research was supported in part by startup research funds from Dartmouth College and by DARPA/NASA subcontract of NCC2-560. Thanks to Carla Ellis, Rick Floyd, and Mike del Rosario.

Introducing Multi-structured File Naming into Unix

Michael McClennen
Stuart Sechrest
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48103

Researchers in the past few years have been applying elements of attribute-based naming not only to resources and services, but to file systems as well [1, 2]. Naming schemes involving non-hierarchical elements can represent more information about files than is possible in strictly hierarchical name spaces, and can solve some of the frustrating limitations of today's file systems. Information stored in existing file systems is often poorly characterized and frequently gets misplaced; for the most part we rely on human memory as an integral component in our organization of computer-based data. These problems are becoming worse as file systems are used to store larger amounts of heterogeneous information for longer periods of time, and as files are shared among different people with divergent strategies of information organization. In these circumstances, the name space must capture more information about files; that is, it must be made more effective at modeling real world properties of files.

A hierarchical name such as *projects/multi-struct/papers/ short.tex* can be considered to be both a *characterization* when the file is created and a *query* when the file is retrieved. In a hierarchical name space, each directory can be thought of as an attribute attached to all of the files in the corresponding subtree. Typically, such an attribute represents some real property common to all of these files. Requiring strictly hierarchical relationships among these properties forces arbitrary decisions as to their relative importance; if the category *papers* is logically orthogonal to the project name *multi-struct*, why should not the name *projects/papers/multi-struct/short.tex* be an equally valid way to call up this file? The hierarchical naming model imposes additional constraints on both characterization and retrieval by limiting the effectiveness of the name space at describing the objects being named. The more components that make up a name, the harder it is to remember them all in order to retrieve the named entity. A hierarchical name space that better characterizes its files becomes harder to use. This is exactly the opposite of what one wants.

The work done by Gifford *et al.* [1] and by Mogul [2] both represent attempts to alleviate these difficulties by developing file services based upon name spaces that are not strictly hierarchical. Each of these systems presents a naming model that allows name components to represent attributes that are not part of the base hierarchy. These non-hierarchical attributes may be used under relaxed ordering constraints in a name presented to the file system for retrieval, or they may be left out entirely. In this way, the information they carry does not place an additional burden upon the user. This kind of approach can dramatically increase the power and flexibility of file naming. However, each of these systems presents only a single alternative to traditional naming. Attributes must belong either to a single hierarchy or to a flat space of attribute labels orthogonal to the hierarchy.

We believe that to realize the full flexibility of semi-hierarchical naming, users must be offered multiple naming constructs from which they can choose the ones that best fit their needs. We have taken the following approach to this problem: we treat hierarchical name spaces as a special case of attribute-based name spaces, with constraints placed on attribute values and on the interpretation of attribute labels in names. We abstract the behaviors of hierarchy into four classes of rules, and

use these rules in systematic ways to produce a set of naming constructs that can be used for file system organization.

A traditional name space is composed of a set of attributes (directories) plus a set of hierarchical relationships. A name space under our system is composed of a set of attributes plus a set of rules from the four classes that define the structure; be it hierarchical, flat, or some combination of other naming constructs. A single directory server that is capable of interpreting these rules can thus manage any name space defined under this framework. We refer to this approach as *multi-structured naming.* Our approach borrows some ideas of data modeling from relational databases, but must meet a different set of requirements. Information stored in file systems constantly changes, requiring dynamic schema. File systems do not make a distinction between schema designer and user; therefore the system interface must remain straightforward and intuitive. A more complete discussion of our system, including a list of naming constructs that we think are useful, may be found in [3].

One of our goals in designing this framework was to make it work with existing computing environments. For better or for worse, no one is likely to use an alternative file naming scheme that requires them to rewrite or relink their applications and utilities, or even one that requires them to use a different command suite to manipulate the file system. As you might guess, the Unix interfaces to the file system are dependent in several ways on the structure of the name space. The following questions are crucial to our research:

- Can non-hierarchical and semi-hierarchical file naming schemes be integrated with the conventional Unix environment?
- What changes must be made to the operating system/utility programs/user programs?
- What compromises to the naming scheme must be made for the sake of compatibility?

We have used our framework of multi-structured naming to implement a prototype file server capable of exporting a variety of semi-hierarchical name spaces, using the NFS protocol. While the prototype is not sufficiently developed to provide a fair assessment of the performance of such a system compared to standard name space implementations, its development has given us insight into the dependencies of Unix and its standard tools on the hierarchical structure of file names.

As a result of our experimentation with this server, we have concluded that multi-structured naming can indeed be integrated into the conventional Unix environment. Most of the changes occur on the server side; the client does not need any kernel modifications, and only a few changes to the standard command suite. The user-level view of the alternative name spaces is based upon a close analog of the existing notion of "directory". Upon completion of a more stable version of our system, we plan to make it available to our colleagues in order to demonstrate in a real-world environment the feasibility and benefits of flexible file naming schemes.

References

[1] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, Pacific Grove CA (USA), October 1991. ACM.

[2] J. C. Mogul. Representing information about files. Report No. STAN-CS-86-1103, Department of Computer Scince, Stanford University, Stanford, California, March 1986.

[3] Stuart Sechrest and Michael McClennen. Blending hierarchical and attribute-based file naming. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992.

The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

* fostering innovation and communicating research and technological developments,
* sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems
* providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter :login:, and a refereed technical quarterly, Computing Systems. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the USENIX Association are:

Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738

ISBN 1-880446-43-X